HIDDEN FALLACIES IN FORMALLY VERIFIED SYSTEMS

A Thesis
Presented to
The Academic Faculty

By

Jan Bobek

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science

Georgia Institute of Technology

May 2020

HIDDEN FALLACIES IN FORMALLY VERIFIED SYSTEMS

Approved by:

Dr. Taesoo Kim, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Dr. Brendan Saltaformaggio
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Calton Pu
School of Computer Science
Georgia Institute of Technology

Date Approved: April 23, 2020

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Taesoo Kim, for suggesting the topic of my thesis and guiding me throughout the thesis writing process, and his students, Meng Xu and Seulbae Kim, who have provided assistance and additional information about Hydra.

I would also like to express my gratitude to Dr. Ada Gavrilovska, Dr. Brendan Saltaformaggio and Dr. Calton Pu, from all of whom I have had the unforgettable chance to learn, for agreeing to serve on my thesis reading committee.

TABLE OF CONTENTS

LIST OF FIGURES

SUMMARY

Formal verification or formal methods represent a rising trend in approaches to correct software construction, i.e. they help us answer the question of how to build software that contains no errors, colloquially known as "bugs." They achieve their goal by providing means for stating theorems about the program under test, and for proving such theorems by methods well-known in mathematics, specifically in mathematical logic. Of course, formal methods are no silver bullet and come with their own set of limitations, the most significant of which is extremely difficult scalability with software size. In spite of the limitations, there have been important breakthroughs in their applications over the last 10–15 years, e.g. Leroy's CompCert [1] (verified C compiler) or Klein's seL4 [2] (verified implementation of the L4 microkernel).

However, how bug-free is verified software in reality? Formal methods make a bold claim that there are indeed no bugs in verified software, or more formally, that the software precisely implements its specification. Unfortunately, as an empirical study from 2017 by Fonseca et al. [3] shows, it may be just too easy to introduce errors into the specification itself, either in form of mistakes ("specification bugs"), or in form of unanticipated assumptions. This thesis aims to take a broader look at formally verified software and formal verification systems, and identify the most common problems in formal methods' applications, leading to bugs still being present in verified software. In particular, the main contribution of this thesis is an overview of several software projects employing formal methods at their core; an empirical study of "real-world" guarantees that the formal verification systems afford them; and, consequently, showcases of different approaches to verified software, along with their strengths and weaknesses.

We believe that understanding how formal methods succeed and fail (or rather, how they can be misused) will be helpful in determining when they become an attractive and worthwhile choice for more ordinary (as opposed to top mission-critical) software. Indeed,

we hope that this thesis may serve as an introductory guide for new projects to the guarantees provided by formal verification; correctness guarantees much stronger than those given by software testing. We hope that in long-term, entry barrier to formal verification will become sufficiently low for formal methods to enter mainstream software development, making developers more confident about their programs, and hopefully ridding our world of "buggy" software once and for all.

CHAPTER 1
INTRODUCTION

Building correct software can be notoriously difficult for humans, and as complexity of the system under construction grows, bugs seem to be inevitable. On the other hand, the cost of incorrect (a.k.a. "buggy") software can be incredibly high in some application domains: the most obvious examples are applications where human lives are at stake, like military, aviation, or healthcare.

Formal verification or *formal methods* represent a rising trend in approaches to correct software construction, i.e. they help us answer the question of how to build software that contains no errors, colloquially known as "bugs." The idea behind formal verification is best illuminated when contrasted with more traditional technique for software quality assurance: software testing. While software testing can assure us that the program under test behaves correctly for certain inputs (*test cases*), it can fundamentally never give us the same guarantee for all inputs; the only way to do it would be to exhaustively test all possible inputs, and even the simplest programs have infinite input spaces (short of memory limitations). There is a beautiful analogy to such an approach in mathematics: software testing amounts to stating a proposition (e.g. "All numbers are even.") and then proceeding to "prove" this proposition by giving examples for which the proposition holds ("2 is even. 10 is even. 42 is even."). Of course, it is beyond obvious that the proposition may still be false; it suffices to give a single counterexample. However, how do we find it? And therein lies the problem with software testing: much like we may never think of odd numbers which reveal the proposition to be false, we may never think of exactly those inputs which cause the program under test to malfunction.

Formal verification takes the other approach: rather than checking a collection of known-good examples, we attempt to prove that no counterexamples exist. Unfortunately, such

effort is almost invariably (much) more difficult than simple testing. At high level, there are two ways to go about trying to prove such a claim: we may rest assured there are no counterexamples either because we have checked all possible cases which might have been a counterexample, or we gave a (hopefully correct) argument for why such a counterexample cannot exist. Observe that "checking all possible cases" may be regarded as easy from a certain perspective: if there is only a finite number of cases to check, it could be done by a machine with some description of all the possible cases and a trusted oracle that always gives the right answer. On the other hand, giving a correct argument for non-existence of the counterexample is much harder and cannot *fundamentally* be done by a computer alone; this is the only way to go when the set of all possible cases is intractably or infinitely large.

1.1 Model checkers and automated verifiers

Model checkers and *automated verifiers*, such as L. Lamport's TLA$^+$ [4] or Z3 Theorem Prover by Microsoft Research [5], respectively, are practical examples of the first, machine-compatible solution. Both accept some sort of specification of what we are trying to prove as input, and output the correct decision whether the proposition is true or not. Moreover, both are able to produce a concrete counterexample in case the proposition does *not* hold. This fact alone is regarded as highly useful during development: the developer is essentially handed a test case for which their implementation fails.

Of course, both feature the same set of limitations we have described above: the set of cases to check must be finite and as small as possible, so automated provers and model checkers go to great lengths in an effort to finitize the testing domain and reduce its size as much as possible. Reducing the number of test cases is often critical for practical applications, because it almost invariably grows exponentially fast with system complexity, easily slipping into the region of non-tractable problems. The developer must be aware of this fact, and sometimes simplifications must be made in the models or propositions to make automated verification feasible.

1.2 Interactive theorem provers

On the other hand, *interactive theorem provers* (ITPs) such as Coq [6, 7] or Isabelle/HOL [8], represent the non-automatable approach. Correctness of ITPs is provided by deduction systems built on sound logic, which prevent inference of false propositions. As suggested by the name, interactive theorem provers require human interaction in order to obtain proofs; they are capable of providing the developer with tools necessary for proof construction and checking the proof for correctness. Sometimes they feature automated proof strategies that are able to automatically prove statements of small to medium complexity; however, the most complicated proofs are typically left to humans.

The most critical limitation of ITPs is severe lack of scalability: compared with writing code, more human effort is necessary to get a correct proof, and while machines can help a bit along the way, bulk of the complicated work is usually left to the user.

1.3 About this thesis

The rest of this thesis is divided in the following manner:

- **Chapter 2**. Chapter 2 introduces notation and basic concepts used in formal verification, such as deductive systems, predicate and first-order logic, etc. The second half of the chapter discusses type theory, which is the foundational basis for many interactive theorem provers used today. We omit more in-depth discussion of the theory behind model checkers and automated verifiers, since they often rely on application domain-specific techniques which are out of the scope of this thesis.

- **Chapter 3**. Chapter 3 surveys almost twenty software projects that rely on formal verification in practice. We focus on systems software, and give a brief outline of the formal methods' role in each specific application. We give more detailed presentation of four formally verified file system implementations, which are selected for fuzzing experiments in the following chapter.

- **Chapter 4**. Chapter 4 presents our fuzzing experiments in terms of the testing setup and results. We introduce our novel FUSE test harness which builds on previous work of Kim et al [9] on file system fuzzing and is designed specifically for tests of FUSE-based file systems. In the second part of the chapter, results of the fuzzing experiments are described, plotted and discussed.

- **Chapter 5**. Chapter 5 summarizes the efforts, contributions and results of this thesis, and suggests a possible direction for future work.

CHAPTER 2
BASIC CONCEPTS IN FORMAL METHODS

In Chapter 2, we introduce the theoretical basis for formal methods and program verification. We briefly describe deductive systems, propositional and first-order logic, and mention some extensions of the first-order logic that are interesting from the program verification perspective. The second part of Chapter 2 talks about type theory, a particularly useful theory suitable for reasoning not only about programs, but as it turns out, about arbitrary logical propositions as well. As such, the theory of typed λ-calculus lies at the heart of Coq, one of the most widely used interactive theorem provers in the world today.

2.1 Logic and proof theory

First, we shall focus on the foundations of program verification, which are constituted by mathematical logic and deductive systems. This section is not meant to be exhaustive, but rather serve as a refresher of the most relevant concepts from formal logic and proofs. Our primary source for the content presented in this section is the work of M. Ben-Ari [10], which the reader is advised to consult for a more in-depth discussion of the upcoming topics.

2.1.1 Deductive systems, proof calculus

Before delving into different logics, let us first consider *deductive systems*. Deductive systems put a formal frame around the intuitive notion that logical reasoning amounts to starting with a collection of *axioms* that are assumed true, and then deriving consequences of the axioms using *inference rules*. Therefore, we can say that axioms and inference rules *together* form a deductive system.

In this work, we will rely exclusively on *sequent calculus* based deductive systems.

Notationally, a *sequent* is

$$P_1, P_2, \ldots, P_n \vdash Q_1, Q_2, \ldots Q_m,$$

where $P_1, P_2, \ldots, P_n$ correspond to *assumptions* and $Q_1, Q_2, \ldots, Q_m$ correspond to *conclusions*. We will interpret the sequent as "if all assumptions hold, then at least one of the conclusions holds." Note that $n = 0$ is allowed; such sequents correspond to conclusions which are *unconditionally true* (*tautologies*), such as

$$\vdash A \implies A$$

With traditional interpretation of the logical symbols $\wedge$ and $\vee$, a sequent is equivalent to

$$\bigwedge_{i=1}^{n} P_i \vdash \bigvee_{j=1}^{m} Q_j.$$

or

$$\vdash \left(\bigwedge_{i=1}^{n} P_i \right) \implies \left(\bigvee_{j=1}^{m} Q_j \right).$$

New sequents can be inferred from previous sequents using inference rules, for which we shall use the following notation:

$$\frac{\text{premise } 1 \qquad \text{premise } 2 \qquad \ldots \qquad \text{premise } N}{\text{conclusion}}$$

To use this rule to infer the conclusion, we must first infer premises 1 through N. Observe that *axioms* can be represented intuitively by setting $N = 0$; the following rule permits to infer P at all times:

$$\frac{}{\vdash P}$$

Now we can proceed to introduction of two kinds of deductive systems: *Hilbert-style* systems and *Gentzen-style* systems.

Hilbert-style systems

The characteristic aspect of a Hilbert-style deduction system is inclusion of only a singular inference rule; the rule is typically the *modus ponens* (a.k.a. implication elimination) rule:

$$\frac{\vdash P \implies Q \qquad \vdash P}{\vdash Q},$$

where P, Q are arbitrary formulas. To make up for the lack of inference rules, Hilbert-style deduction systems typically introduce a multitude of *axiom schemes*, which allow for derivation of more complex formulas. Three well-known axiom schemes are

Axiom 1. $\vdash A \implies (B \implies A)$,

Axiom 2. $\vdash (A \implies (B \implies C)) \implies ((A \implies B) \implies (A \implies C))$,

Axiom 3. $\vdash (\neg B \implies \neg A) \implies (A \implies B)$,

where A, B, C stand for arbitrary formulas. However, the choice of fundamental axiom schemes is not unique; in particular, only a single axiom scheme is enough to yield a deduction system just as strong:

Meredith's axiom. $\vdash ((((A \implies B) \implies (\neg C \implies \neg D)) \implies C) \implies E)$
$\implies ((E \implies A) \implies (D \implies A))$.

Consider the following example, in which the formula $A \implies A$ is derived in a Hilbert-style deduction system featuring the Axioms 1–3 listed above above:

$$
\begin{aligned}
&1. \vdash (A \implies ((A \implies A) \implies A)) \\
&\qquad \implies ((A \implies (A \implies A)) \implies (A \implies A)) && \text{(Axiom 2)} \\
&2. \vdash A \implies ((A \implies A) \implies A) && \text{(Axiom 1)} \\
&3. \vdash (A \implies (A \implies A)) \implies (A \implies A) && \text{(MP 1, 2)} \\
&4. \vdash A \implies (A \implies A) && \text{(Axiom 1)} \\
&5. \vdash A \implies A && \text{(MP 3, 4)}
\end{aligned}
$$

As the example shows, working with Hilbert-style deduction systems can be often quite tedious and unintuitive.

Gentzen-style systems

Gentzen-style deduction systems take the opposite approach; such systems typically feature only one axiom scheme, the *complementary pair*:

$$\frac{}{\vdash \Pi, A, \neg A}$$

where A is an arbitrary formula, and Π is a collection of arbitrary formulas (a.k.a. *context*). On the other hand, plenty of inference rules are typically available, e.g.

$$(1)\ \frac{\vdash \Pi_1, A \qquad \vdash \Pi_2, \neg B}{\vdash \Pi_1, \Pi_2, \neg(A \implies B)} \qquad (2)\ \frac{\vdash \Pi, \neg A, B}{\vdash \Pi, A \implies B}$$

The formula $A \implies (\neg B \implies \neg(A \implies B))$ is derived in the following example to demonstrate a Gentzen-style deduction system:

1. $\vdash A, \neg A$	(Axiom)
2. $\vdash \neg B, \neg\neg B$	(Axiom)
3. $\vdash \neg A, \neg\neg B, \neg(A \implies B)$	(Rule 1: 1, 2)
4. $\vdash \neg A, \neg B \implies \neg(A \implies B)$	(Rule 2: 3)
5. $\vdash A \implies (\neg B \implies \neg(A \implies B))$	(Rule 2: 4)

2.1.2 Propositional logic

Propositional logic is the basic formalism which captures the concept of truth of a statement. The set of all propositional formulas $\mathcal{F}$ can be defined recursively as follows:

1. **Atoms.** $\mathcal{P} \subseteq \mathcal{F}$, where $\mathcal{P}$ is an unbounded set of symbols called *atomic propositions* or *atoms*.

2. **Boolean operators.** If $P, Q \in \mathcal{F}$, then

   - $\neg P \in \mathcal{F}$,
   - $P \vee Q \in \mathcal{F}$,
   - $P \wedge Q \in \mathcal{F}$,
   - $P \implies Q \in \mathcal{F}$,
   - $P \iff Q \in \mathcal{F}$,
   - $\ldots$

   The list of Boolean operators is non-exhaustive, additional operators are sometimes used.

Since we wish to include arbitrary propositional formulas in our deduction systems, we need to specify how the system interacts with the Boolean operators. One option is to take advantage of the fact that any Boolean operator can be expressed using implication and negation only, e.g.

$$P \vee Q \equiv \neg P \implies Q,$$
$$P \wedge Q \equiv \neg(P \implies \neg Q).$$

This is a suitable approach for Hilbert-style deduction systems. In Gentzen-style systems, we can introduce additional inference rules instead, e.g.

$$\frac{\vdash \Pi_1, P \qquad \vdash \Pi_2, Q}{\vdash \Pi_1, \Pi_2, P \wedge Q} \qquad \frac{\vdash \Pi, P, Q}{\vdash \Pi, P \vee Q}$$

Ultimately, we wish to investigate truthfulness of the propositions in question, so we need to define semantics for the formulas. Let $A \in \mathcal{F}$ and let $\mathcal{P}_A$ be the set of atoms appearing in A. An *interpretation* for A is a function $\mathcal{I}_A : \mathcal{P}_A \to \{\mathrm{T}, \mathrm{F}\}$, i.e. a function which assign *truth values* T and F to all atoms in A. With $\mathcal{I}_A$ we can define $v_{\mathcal{I}}(A)$, the *truth value of A under $\mathcal{I}_A$*, inductively as follows:

$$v_{\mathcal{I}}(p) = \mathcal{I}_A(p), \quad p \in \mathcal{P}_A \qquad v_{\mathcal{I}}(A \vee B) = \begin{cases} \mathrm{T}, & v_{\mathcal{I}}(A) = \mathrm{T} \text{ or } v_{\mathcal{I}}(B) = \mathrm{T} \\ \mathrm{F}, & \text{otherwise} \end{cases}$$

$$v_{\mathcal{I}}(\neg A) = \begin{cases} \mathrm{F}, & v_{\mathcal{I}}(A) = \mathrm{T} \\ \mathrm{T}, & v_{\mathcal{I}}(A) = \mathrm{F} \end{cases} \qquad v_{\mathcal{I}}(A \wedge B) = \begin{cases} \mathrm{T}, & v_{\mathcal{I}}(A) = \mathrm{T} \text{ and } v_{\mathcal{I}}(B) = \mathrm{T} \\ \mathrm{F}, & \text{otherwise} \end{cases}$$

Truth values of the rest of the Boolean operators can be intuitively defined in a similar manner.

The notion of truth value under an interpretation allows us to define several interesting properties of proposition formulas:

- **Logical equivalence.** Let $A, B \in \mathcal{F}$. If $v_{\mathcal{I}}(A) = v_{\mathcal{I}}(B)$ under an arbitrary interpretation $\mathcal{I}$, we say A is *logically equivalent* to B, denoted $A \equiv B$. Note that $A \equiv B$ if and only if $v_{\mathcal{I}}(A \iff B) = \text{T}$ under any interpretation $\mathcal{I}$. Logical equivalence forms the basis for *substitution*, i.e. if $A \equiv B$, we may replace any occurrence of A with B (and vice versa) without affecting the truth value.

- **Logical consequence.** Let $A, B \in \mathcal{F}$. We say B is a *logical consequence* of A, denoted $A \models B$, if under all interpretations $\mathcal{I}$ such that $v_{\mathcal{I}}(A) = \text{T}$, $v_{\mathcal{I}}(B) = \text{T}$ holds. Note that $A \models B$ if and only if $v_{\mathcal{I}}(A \implies B) = \text{T}$ under any interpretation $\mathcal{I}$.

- **Satisfiability.** Let $A \in \mathcal{F}$. We say A is *satisfiable* if and only if there exists an interpretation $\mathcal{I}$ such that $v_{\mathcal{I}}(A) = \text{T}$. A is *unsatisfiable* if and only if it is not satisfiable, i.e. $v_{\mathcal{I}}(A) = \text{F}$ under any interpretation $\mathcal{I}$.

- **Validity.** Let $A \in \mathcal{F}$. We say A is *valid* (or a *tautology*) if and only if $v_{\mathcal{I}}(A) = \text{T}$ under an arbitrary interpretation $\mathcal{I}$. A is *falsifiable* if and only if it is not valid, i.e. there exists an interpretation $\mathcal{I}$ such that $v_{\mathcal{I}}(A) = \text{F}$.

  Observe that A is valid if and only if $\neg A$ is unsatisfiable, and A is satisfiable if and only if $\neg A$ is falsifiable.

2.1.3 First-order logic

In order to motivate first-order logic, observe that propositional logic is not enough to fully describe statements like

$$4x - 3 > 2y + 8.$$

Indeed, this statement is not absolutely true or false; its truthfulness depends on what exactly is meant by variables x, y, operators $+, -, \cdot$, and the relational operator $>$.

First-order logic is a more-or-less straightforward extension of predicate logic to incorporate the notions of a *domain* $\mathcal{D}$; *functions*, which map elements of $\mathcal{D}$ to other elements of $\mathcal{D}$; and *predicates*, which map elements of $\mathcal{D}$ to truth values T and F. Formally, we recursively define the *set of terms* $\mathcal{T}$:

1. **Constants.** $\mathcal{A} \subseteq \mathcal{T}$, where $\mathcal{A}$ is a countable set of *constant symbols*.

2. **Variables.** $\mathcal{V} \subseteq \mathcal{T}$, where $\mathcal{V}$ is a countable set of *variable symbols*.

3. **Functions.** If $f^n \in \mathcal{F}$ and $t_1, t_2, \ldots, t_n \in \mathcal{T}$, then $f^n(t_1, t_2, \ldots, t_n) \in \mathcal{T}$, where $\mathcal{F}$ is a countable set of *function symbols*; $n \in \mathbb{N}$ is called *arity* of the function f^n (or equivalently, f^n is an n*-ary function*).

With the set of terms $\mathcal{T}$ we are able to recursively define the *set of formulas in first-order logic* $\mathcal{F}$:

1. **Atomic formula.** If $p^n \in \mathcal{P}$ and $t_1, t_2, \ldots, t_n \in \mathcal{T}$, then $p^n(t_1, t_2, \ldots, t_n) \in \mathcal{F}$, where $\mathcal{P}$ is a countable set of *predicate symbols*; $n \in \mathbb{N}$ is called *arity* of the predicate p^n (or equivalently, p^n is an n*-ary predicate*).

2. **Boolean operators.** If $A, B \in \mathcal{F}$, then

   - $\neg A \in \mathcal{F}$,
   - $A \vee B \in \mathcal{F}$,
   - $A \wedge B \in \mathcal{F}$,
   - $A \implies B \in \mathcal{F}$,
   - $A \iff B \in \mathcal{F}$,
   - $\ldots$

This definition is adopted from propositional logic without change.

3. **Quantifiers.** If $x \in \mathcal{V}$ and $A \in \mathcal{F}$, then

   - $\forall x\ A \in \mathcal{F}$,
   - $\exists x\ A \in \mathcal{F}$.

Note that at high level, there are two significant differences from propositional logic: first, the notion of atoms was extended into atomic formulas with their own recursive set of terms; and second, the concept of quantifiers was introduced.

As in the case of Boolean operators, we need to define how deductive systems interact with the quantifiers if we hope to to use them to manipulate formulas in first-order logic. Recall that any $\exists$ operator may be turned into a $\forall$ operator using the identity

$$\exists x\ A(x) \equiv \neg \forall x\ \neg A(x)$$

In Hilbert-style systems, two new axioms for manipulating $\forall$ formulas can be introduced:

Axiom 4. $\vdash \forall x\ A(x) \implies A(a)$,

Axiom 5. $\vdash \forall x\ (A \implies B(x)) \implies (A \implies \forall x\ B(x))$,

in addition to second inference rule, known as *generalization*:

$$\frac{\vdash A(a)}{\vdash \forall x\ A(x)}$$

In Gentzen-style deduction systems, we can simply add inference rules such as

$$\frac{\vdash \Pi_1, \exists x\ A(x), A(a)}{\vdash \Pi_1, \exists x\ A(x)} \qquad \frac{\vdash \Pi_2, A(a)}{\vdash \Pi_2, \forall x\ A(x)}$$

Note that in the second inference rule, a must not occur in any formula in Π_2.

The concepts familiar from propositional logic, such as interpretation, logical equivalence and consequence, satisfiability and validity can be extended over to first-order logic while maintaining their intuitive meanings; the definitions are largely technical and we omit them from our work for brevity. Interested readers are advised to consult our primary source [10].

2.1.4 Beyond first-order logic

Higher-order logic

It is possible to extend first-order logic further, similarly to extending the propositional logic, to obtain higher-order logics[1] [11]. Higher-order logic permits predicates which take other predicates as parameters, or allows quantifiers to quantify over predicates or functions. Note that in first-order logic, an n-ary predicate can be thought of as a subset of $\mathcal{D}^n$ (the set of tuples with n elements from $\mathcal{D}$); in higher-order logic, these sets become powersets, i.e. sets of sets. While interesting from a theoretical perspective, many useful algorithms that exist for first-order logic (e.g. a proof-checking algorithm) do not exist or are inefficient for higher-order logics, which severely limits their practical applications.

Intuitionistic logic

Intuitionistic logic [12] is an interesting restriction of classical logic which follows the notion of a constructive proof more closely. For example, the informal semantic meaning of a formula $A \wedge B$ is not that A is true *and* B is true, but rather that one is able to supply *a proof of* A *and a proof of* B. In a similar spirit, the formula $A \implies B$ is considered to consist of a *method of converting* any proof of A into a proof of B.

This fairly simple restriction has far-reaching consequences; most importantly, some tautologies which are taken for granted in classical logic, such as the law of excluded middle ($A \vee \neg A$), double-negation elimination ($\neg\neg A \iff A$) or proof by contradiction

[1] Note that propositional logic is sometimes called the *zeroth-order logic*.

$((\neg A \implies \bot) \implies A)$, cannot be proved (and hence used) in intuitionistic logic. Note that this position makes sense from a certain perspective, e.g. a proof by contradiction only shows that assuming $\neg A$ leads to a contradiction (as opposed to demonstrating A directly), and is therefore an intrinsically non-constructive proof.

However, associating a concrete proof with every proved formula has certain attractive advantages, too. In particular, the Coq interactive theorem prover is able to *automatically extract programs* from proof terms written in intuitionistic logic. Automatic code extraction from proofs is a very useful feature depended on by several software projects which use Coq for formal verification (more details are given in Chapter 3).

2.1.5 Extensions of logic for computer programs

Hoare logic

In his seminal paper from 1969 [13], a mathematician C. A. R. Hoare introduced an axiomatic basis for computer programming, colloquially referred to as Hoare logic. The novel concept in Hoare logic are so-called *Hoare triples*, which are statements of the form

$$P\ \{Q\}\ R,$$

where P and R are assertions in the underlying logic, named *pre-condition* and *post-condition*, respectively, and Q is a (fragment of) computer program. Proposed semantics for a Hoare triple is: if the statement P holds immediately before executing Q, then R will hold after Q runs to completion.

To manipulate Hoare triples in a deductive system, new inference rules or axioms must be introduced; for brevity, we only consider Gentzen-style deductive systems. The most trivial inference rules are named *strengthening the pre-condition* and *weakening the post-*

condition, for obvious reasons:

$$\frac{\vdash \Pi_1, P\ \{Q\}\ R \qquad \vdash \Pi_2, S \implies P}{\vdash \Pi_1, \Pi_2, S\ \{Q\}\ R} \qquad \frac{\vdash \Pi_1, P\ \{Q\}\ R \qquad \vdash \Pi_2, R \implies S}{\vdash \Pi_1, \Pi_2, P\ \{Q\}\ S}$$

Elementary programs can be assigned Hoare triples by axioms. For example, consider the *assignment axiom*:

$$\frac{}{\vdash \Pi, P'\ \{x := E\}\ P}$$

where P' is obtained from P by substituting all occurrences of x in P with E.[2]

Hoare triples describing larger fragments of programs may be obtained via composition using the so-called *composition rule*:

$$\frac{\vdash \Pi_1, P\ \{Q_1\}\ R \qquad \vdash \Pi_2, R\ \{Q_2\}\ S}{\vdash \Pi_1, \Pi_2, P\ \{Q_1; Q_2\}\ S}$$

where the ; operator in $Q_1; Q_2$ is the sequential composition operator for programs, familiar from languages like C. Intuitively, execution of the program $Q_1; Q_2$ consists of execution of Q_1 immediately followed by execution of Q_2.

The most involved inference rule given by Hoare is the *rule of iteration*, which allows to infer pre- and post-condition of a `while` loop:

$$\frac{\vdash \Pi, P \wedge B\ \{S\}\ P}{\vdash \Pi, P\ \{\texttt{while}\ B\ \texttt{do}\ S\}\ \neg B \wedge P}$$

P is the so-called *loop invariant*. When using the rule of iteration, we must first show that the body of the loop S re-establishes the loop invariant P, given that both the invariant and the loop condition B hold. Then, the rule of iteration allows us to infer that if P holds before entering the loop, P will also hold at the end of the loop *in addition* to the loop condition being false. Note that the rule of iteration has a built-in assumption of termination: the rule allows to infer $\neg B$, which is a contradiction for infinite loops. However, Hoare logic is still

[2]Note that E must not have any side-effects.

sound: the triple $P\ \{Q\}\ S$ asserts that S is established after Q *runs to completion*, which never occurs with an infinite loop.

Separation logic

Separation logic, described by J. C. Reynolds in 2002 [14], allows reasoning about programs in presence of heaps, pointers, and dynamically allocated objects. The key innovation lies in extending Hoare logic with assertions about heaps:

emp	empty heap
$t_1 \mapsto t_2$	singleton heap
$A * B$	separating conjunction
$A \mathrel{-\!\!*} B$	separating implication

where $t_1, t_2 \in \mathcal{T}$ (terms) and $A, B \in \mathcal{F}$ (formulas). The first assertion (*empty heap*) asserts that the heap is empty; the second assertion (*singleton heap*) says the heap contains precisely one mapping, which maps an address t_1 to contents t_2; the third assertion (*separating conjunction*) claims that the heap can be split into two disjoint parts in which A and B hold, respectively; and finally, the fourth assertion (*separating implication*) means that if the heap is extended with a disjoint heap in which A holds, then B will hold in the extended heap. Naturally, the programming language that is being reasoned about must incorporate elements for manipulating dynamic memory; Reynolds uses the following notation:

$x := \mathbf{cons}(E_1, \ldots, E_n)$	allocation
$x := [E_1]$	lookup
$[E_1] := E_2$	mutation
$\mathbf{dispose}(E_1)$	deallocation

where $E_1, \ldots, E_n$ are *program expressions*. Moreover, semantics of the programs must include notion of the heap, i.e. a mapping from addresses to values that may be mutated throughout execution of the program.

The intuitive ideas behind separation logic can be illustrated using its inference rules. The relationship between separating conjuction and separating implication is described by the following two inference rules:

$$\frac{\vdash \Pi, A * B \implies C}{\vdash \Pi, A \implies (B \mathrel{-\!\!*} C)} \qquad \frac{\vdash \Pi, A \implies (B \mathrel{-\!\!*} C)}{\vdash \Pi, A * B \implies C}$$

Empty heap and singleton heap assertions are best shown in the axioms of separation logic, which define how these assertions interact with the corresponding program elements. For *(local, non-interfering) allocation*, we have

$$\frac{}{\vdash \Pi, \textbf{emp}\ \{x := \textbf{cons}(E)\}\ x \mapsto E}$$

where x is not free in E (the non-interference[3] property). Similarly, the *(local) deallocation axiom* is

$$\frac{}{\vdash \Pi, \exists x\ E \mapsto x\ \{\textbf{dispose}(E)\}\ \textbf{emp}}$$

Finally, the *(local) mutation axiom* is defined as

$$\frac{}{\vdash \Pi, \exists x\ E_1 \mapsto x\ \{[E_1] := E_2\}\ E \mapsto E_2}$$

Last but definitely not least, note that all the inference rules above are *local*, i.e. no other statements hold simultaneously. To lift the local inference rules into global rules, a

[3]This is not a fundamental restriction of separation logic; a more complex definition may be given which allows x to freely occur in E. However, we give the uncomplicated definition for simplicity.

frame rule is introduced:

$$\frac{\vdash \Pi, P\ \{Q\}\ R}{\vdash \Pi, P * S\ \{Q\}\ R * S}$$

where no variable occurring free in S (*the frame*) is modified by Q. Intuitively, the frame rule asserts that we may introduce arbitrary additional constraints, provided they are completely irrelevant with respect to the program Q.

2.2 Type theory

At the turn of 20th century, B. Russell famously demonstrated that formalizations of the naïve set theory put forth by G. Cantor some years earlier lead to paradoxes. Probably the most well-known paradoxical statement involves a set S containing all sets which are *not members of themselves*:

$$S = \{x \mid x \notin x\}.$$

The paradox arises when one attempts to analyze truthfulness of the statement $S \in S$; indeed, according to the law of excluded middle, either $S \in S$ or $S \notin S$ is true, but assuming one immediately leads to a proof of the other, i.e. a contradiction.

Mathematicians and logicians attempted to address this paradox in different ways. One line of work focused on refining axioms of the set theory so that this paradox can be avoided, and eventually evolved into Zermelo-Fraenkel (ZF) and Zermelo-Fraenkel with the Axiom of Choice (ZFC) set theory, by far the most mainstream set theory used today.

Russell himself, however, went down the other path, attempting to restrict the logical language in which the paradox is stated. That is perhaps the more intuitive approach: the statement $x \in x$ feels "wrong," since intuitively x can either be an object or a set of x-like objects, but not both at the same time. The restriction of such statements led to development of *type theory*.

Amazingly, type theory has many applications in programming languages. As we shall

see in the following subsection, there is a structurally equivalent paradox of *self-application* in untyped λ-calculus; however, this dual paradox may be regarded as only a tip of the iceberg. There is a deep and fundamental link between logic and programs, known as the *Curry-Howard correspondence* or *isomorphism*, which states that types may be interpreted as *propositions*, and (typed) programs then become *proofs* of the corresponding propositions. This surprising development is summarized in the rest of the chapter; a book by R. Nederpelt and H. Geuvers [15] shall be our elementary source, to which the reader is referred for details.

2.2.1 Untyped λ-calculus

λ-calculus is a formalism introduced by A. Church in 1936 [16] in an attempt to describe what functions can be computed. Despite its power, basic λ-calculus is remarkably simple and intuitive; the recursive set of λ-calculus terms, denoted Λ, can be defined as follows:

1. **Variables.** $\mathcal{V} \subseteq \Lambda$, where $\mathcal{V}$ is a countable set of *variable symbols*.
2. **Abstraction.** If $x \in \mathcal{V}$ and $A \in \Lambda$, then $\lambda x.\ A \in \Lambda$.
3. **Application.** If $A, B \in \Lambda$, then $MN \in \Lambda$.

Abstraction and application may be intuitively regarded as an (anonymous) function declaration and function invocation. The following example demonstrates the idea behind representing a computation using λ-calculus:

$$\begin{aligned} (\lambda x.\ 3x - 5)7 \quad &\rightarrow \quad 3 \cdot 7 - 5 \\ &\rightarrow \quad 16. \end{aligned}$$

Note that integer arithmetic is *not* part of λ-calculus definition; however, it is possible to find λ-calculus terms for operators $+, \cdot$ and integers $1, 2, \ldots$.[4] so that arithmetic with these

[4] Such terms are called *Church numerals.*

terms behaves as expected.

Several straightforward properties of λ-calculus terms can de defined:

1. **α-equivalence.** Two λ-calculus terms are called *α-equivalent* if and only if they are the same modulo renaming variables bound by abstractions. α-equivalence formalizes the intuitive notion that the two terms $\lambda x.\ x + 5$ and $\lambda y.\ y + 5$ should be equivalent, since the particular name of the variable introduced by abstraction does not "matter."

2. **β-reduction.** The concept of β-reduction formally captures the notion of computation, hinted at in the example above. A single step of β-reduction eliminates one abstraction and one application from the term, substituting the second term in the application into the body of the abstraction wherever the abstraction variable occurs.

3. **β-equivalence.** Two terms $M, N \in \Lambda$ are *β-equivalent* if there exist two series of β-reductions (one for each term) which reduce M, N to the same term.

4. **β-normal form.** A λ-calculus term is in β-normal form if we cannot apply any β-reduction steps. In the example above, 16 is considered to be in β-normal form; in a certain sense, the β-normal form represents the "result" of the computation.

5. **Normalization.** A term $M \in \Lambda$ is *weakly normalizing* if there exists a term $N \in \Lambda$ such that M and N and β-equivalent and N is in β-normal form. The term M is *strongly normalizing* if there is no infinite series of β-reductions applicable to M. Note that strong normalization implies weak normalization.

On the positive side, untyped λ-calculus is as computationally powerful as it can be; it has been proven equivalent to Turing machines and other general models of computation [17], despite its conceptual and notational simplicity. On the negative side, desirable properties such as guaranteed strong normalization are missing. This is not only a theoretical restriction; in fact, there is a λ-calculus term $Y \in \Lambda$, called the *fixed-point combinator*,

which constructs an infinitely β-reducible term for an arbitrary λ-calculus term:

$$Y \equiv \lambda y.\ (\lambda x.\ y(xx))(\lambda x.\ y(xx))$$

Indeed, given an arbitrary term $L \in \Lambda$, β-reducing the term YL produces a term which has YL as a subterm, thus guaranteeing infinite series of possible β-reductions:

$$\begin{aligned} YL \quad &\to \quad (\lambda x.\ L(xx))(\lambda x.\ L(xx)) \\ &\to \quad L((\lambda x.\ L(xx))(\lambda x.\ L(xx))) \quad \leftarrow \quad L(YL) \end{aligned}$$

Note that the fixed-point combinator Y relies on self-application xx, whose meaning is unclear and unintuitive (similarly to $x \in x$ in Russell's paradox).

2.2.2 Simply-typed λ-calculus ($\lambda^{\to}$)

Four years later (in 1940), A. Church himself presented an extension of λ-calculus [18] which incorporates some of B. Russell's ideas about types, addressing the outlined shortcomings of pure λ-calculus.

The *set of types* $\mathbb{T}$ can be defined as follows:

1. **Type variables.** $\{\sigma, \tau, \ldots\} \subseteq \mathcal{V}_T \subseteq \mathcal{T}$, where $\mathcal{V}_T$ is a countable set of *type variables*.
2. **Function types.** If $\sigma, \tau \in \mathbb{T}$, then $\sigma \to \tau \in \mathbb{T}$.

The typing judgements are typically written in terms of a *context* Γ, a term $M \in \Lambda$ and a type $\sigma \in \mathbb{T}$. The assertion

$$\Gamma \vdash M : \sigma$$

is interpreted as "assuming the typing judgements in Γ hold, the term M has the type σ." The two inference rules for abstraction and application in simply-typed λ-calculus $\lambda^{\to}$ are

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.\ M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

in addition to the trivial axiom

$$\frac{}{\Gamma \vdash x : \sigma} \qquad \text{if } x : \sigma \in \Gamma$$

Note that the context Γ only holds typing judgements of the form $x : \sigma, x \in \mathcal{V}, \sigma \in \mathbb{T}$.

There are three questions with regards to types that we might desire an answer for:

1. **Well-typedness.** Given a term $M \in \Lambda$, find context Γ and a type $\sigma \in \mathbb{T}$ such that $\Gamma \vdash M : \sigma$.

2. **Type checking.** Given a context Γ, a term $M \in \Lambda$ and a type $\sigma \in \mathbb{T}$, decide whether $\Gamma \vdash M : \sigma$.

3. **Term finding.** Given a context Γ and a type $\sigma \in \mathbb{T}$, find a term $M \in \Lambda$ such that $\Gamma \vdash M : \sigma$.

As it turns out, each of these problems is *decidable* in $\lambda^{\to}$, i.e. there exists an algorithm which produces the correct answer. This stands in stark contrast with more powerful typed λ-calculus systems that we describe in the following subsections; in those systems, at least some of these problems are usually undecidable. Term finding is typically the culprit, since under the Curry-Howard isomorphism, the term finding problem actually asks for a proof of an arbitrary proposition.

Restricting the set of terms to *typeable* terms (i.e. only those which have a type) guarantees useful properties of $\lambda^{\to}$: by *uniqueness of types* we have a unique type for each term, which can be shown to rule out self-application and the fixed-point combinator; *strong normalization* guarantees that every term has a (unique) β-normal form without the risk of infinite chain of β-reductions. These properties come at a high price, though, because $\lambda^{\to}$ lacks significant portion of the computational expressivity of untyped λ-calculus. In fact, it is possible to define Church numerals, addition and multiplication, but not much else. Therefore, the following systems will attempt to recover some of the computational power

by using more sophisticated type systems.

2.2.3 System F

System F, a.k.a. the second-order λ-calculus $\lambda 2$, was introduced by J.-Y. Girard in his dissertation from 1972; his own account of System F and its impact on the subsequent research can be found in a paper published in 1986 [19]. Although Girard's reasons for studying System F were more theoretical, it is possible to motivate the extensions introduced in System F by the following question: what should be the type of the identity function? By identity function, we mean the λ-calculus term $\lambda x.\ x$.

In $\lambda^{\to}$, the identity function can be typed as $\vdash \lambda x : \sigma.\ x : \sigma \to \sigma$. However, note that the type variable σ is in fact part of the term itself: $\lambda x : \sigma.\ x$ literally includes the type variable σ. In practice, this means we have a separate identity function for each type:

$$\vdash \lambda x : \text{nat}.\ x : \text{nat} \to \text{nat}$$

$$\vdash \lambda x : \text{bool}.\ x : \text{bool} \to \text{bool}$$

$$\vdash \lambda x : \text{nat} \to \text{bool}.\ x : (\text{nat} \to \text{bool}) \to (\text{nat} \to \text{bool})$$

What we might seek instead is a *polymorphic* identity function, i.e. a single function which works for all possible types. In System F, such a function can be defined by the term

$$\lambda \alpha : *.\ \lambda x : \alpha.\ x$$

Note that the syntax includes another abstraction, but on the *type level*: the abstraction introduces a new type variable α, whose "type" is $*$. This observation hints at a different perspective when classifying the typed λ-calculus systems: in $\lambda^{\to}$, both the abstracted variable and the body of the abstraction could only terms, i.e. the abstraction was a *term depending on term*. In System F, we can also have term abstractions which introduce a type variable, i.e. System F permits *terms depending on types*.

Immediately, we need to answer another question: what is the *type* of such a polymorphic function? As it turns out, the type variable α must be introduced into the type as well, using a new construct called the Π-*type*:

$$\vdash \lambda\alpha : *.\ \lambda x : \alpha.\ x : (\Pi\alpha : *.\ \alpha \to \alpha)$$

These two extensions combined are enough to provide the kind of polymorphism we set out to achieve with our identity function example.

To define the *set of types* $\mathbb{T}$ used in System F, we will extend the definition of $\mathbb{T}$ from $\lambda^{\to}$ (given in Subsection 2.2.2) with a third item:

3. Π**-types.** If $\alpha \in \mathcal{V}_T$ and $A \in \mathbb{T}$, then $\Pi\alpha : *.\ A \in \mathbb{T}$.

Similarly, we extend the $\lambda^{\to}$'s definition of *set of terms* Λ with two new items:

4. **Second order (type) abstraction.** If $\alpha \in \mathcal{V}_T$ and $M \in \Lambda$, then $\lambda\alpha : *.\ M \in \Lambda$.

5. **Second order (type) application.** If $A \in \mathbb{T}$ and $M \in \Lambda$, then $MA \in \Lambda$.

In the deduction system for typing, the two most important additions are the inference rules which define how to deal with types of terms involving second order abstraction and application:

$$\frac{\Gamma, \alpha : * \vdash M : A}{\Gamma \vdash \lambda\alpha : *.\ M : (\Pi\alpha : *.\ A)} \qquad \frac{\Gamma \vdash M : (\Pi\alpha : *.\ A) \qquad \Gamma \vdash B : *}{\Gamma \vdash MB : A\,[\alpha := B]}$$

where $A\,[\alpha := B]$ denotes the type A in which all occurrences of the type variable α have been replaced by the type B.

In terms of properties, all proofs about the "nice" behavior of $\lambda^{\to}$ terms (e.g. strong normalization) can be extended to System F; unfortunately, some of the algorithmic typing problems become more difficult. For example, System F permits self-application, as evidenced by the typeable System F term $\lambda x : (\Pi\alpha : *.\ \alpha \to \alpha)\, x(\sigma \to \sigma)(x\sigma)$. On the

other hand, the fixed-point combinator-like term $(\lambda x.\ xx)(\lambda x.\ xx)$ is *not* typeable, but that is far from obvious. A well-typedness algorithm would be able to provide answers to such questions, but in 1994 J. B. Wells proved that both well-typedness and type checking are undecidable in System F [20]. That being said, certain restrictions of the notion of polymorphism not only allow both problems to remain decidable, but even efficiently solvable [21].

2.2.4 System Fω

System F introduces polymorphic terms, i.e. terms whose type involves a fresh type variable. Analogically, we might want to construct *types* which include a fresh type variable, that is, *types depending on types*; that is what System Fω, an extension of System F also introduced by J.-Y. Girard, achieves. For a concrete motivational example, consider a function computing length of a polymorphic list: the function takes a list of values of type α and returns a natural number:

$$\lambda\alpha : *.\ \lambda x : \text{list}\ \alpha.\ (\ldots) : \Pi\alpha : *.\ \text{list}\ \alpha \to \text{nat}$$

Note that the list is *polymorphic*: a list of α-typed values has the type list α. This raises an important question: what is the "type" of list? We may observe that list takes a single type $\alpha : *$ as a parameter, and is in itself a type, i.e. list $\alpha : *$, which invariably leads us to conclude that the "type" of list must be

$$\text{list} : * \to *,$$

which is the "type" of a *type constructor*: given a type, the type constructor returns another type. Note that we put the word "type" in quotation marks; indeed, calling terms like

$$*, * \to *, (* \to *) \to *, (* \to *) \to (* \to *), \ldots$$

"types" is imprecise at best. We adopt the name *kinds* instead, in order to differentiate between such terms and proper types. We also define the *type of all kinds* to be $\Box$, i.e. for any kind κ, $\kappa : \Box$. Finally, a *sort* is defined to mean either $*$ or $\Box$.

To formalize the extensions outlined above, first we need recursively define the *set of kinds* $\mathbb{K}$:

1. **Atomic type.** $* \in \mathbb{K}$.

2. **Type constructor.** If $A, B \in \mathbb{K}$, then $A \to B \in \mathbb{K}$.

Next, the definition of the *set of types* $\mathbb{T}$ given in Subsection 2.2.3 is adjusted to account for more than one kind:

3. Π**-types.** If $\alpha \in \mathcal{V}_T$, $A \in \mathbb{K}$ and $B \in \mathbb{T}$, then $\Pi\alpha : A.\ B \in \mathbb{T}$.

In the definition of the *set of terms* Λ, we need to change the second order abstraction in a similar way:

4. **Second order (type) abstraction.** If $\alpha \in \mathcal{V}_T$, $A \in \mathbb{K}$ and $M \in \Lambda$, then $\lambda\alpha : A.\ M \in \Lambda$.

The inference rules for abstraction and application are modified analogically:

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.\ B : s}{\Gamma \vdash \lambda x : A.\ M : (\Pi x : A.\ B)} \qquad \frac{\Gamma \vdash M : (\Pi x : A.\ B) \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B\,[x := N]}$$

where s is a sort. Note that there are some technicalities to be taken care of; namely, we identify

$$\sigma \to \tau \equiv \Pi x : \sigma.\ \tau,$$

and there needs to be another premise in the abstraction rule. An interested reader is advised to consult [15] for details.

2.2.5 λP

System F and System Fω extend $\lambda^{\to}$ with terms depending on types and types depending on types, respectively; λP introduces the last combination, i.e. *types depending on terms*, also known as *dependent types*. As a quick motivation, consider V $n : *$ (with n : nat), the type of vectors of length n:

$$\begin{aligned} &(1,2,3),(-30,21,18),\ldots && : \text{V } 3 \\ &(42),(108),(-37),\ldots && : \text{V } 1 \\ &(1,2,\ldots,n-1,n) && : \text{V } n \end{aligned}$$

In all of the examples above, the *type constructor* V has the type nat $\to *$, i.e. given a natural number, V constructs a type.

This extension can be achieved by yet again introducing the Π-types into the definition of the set of all types $\mathbb{T}$ given in Subsection 2.2.3; however, this time the argument must be a term:

3. Π**-types.** If $x \in \mathcal{V}$, $A \in \mathbb{T}$ and $B \in \mathbb{T}$, then $\Pi x : A.\ B \in \mathbb{T}$.

On the other hand, the definition of the set of all terms Λ remains the same as in Subsection 2.2.2, and the abstraction and application inference rules are identical to what was presented in Subsection 2.2.4.

An attractive property of λP is its ability to directly encode arbitrary propositions and proofs in the types and terms, as the Curry-Howard isomorphism states:

1. **Propositions.** We may look at an arbitrary type $P : *$ as a *proposition*, which is *inhabited* (i.e. there exists a term of type P) if and only if P can be proved; moreover, terms $x : P$ encode the *proof of* P.

2. **Sets.** Similarly, we may see that sets can also be encoded as types, i.e. for a set S we

have $S : *$, and all elements $a \in S$ correspond to terms $a : S$. If S is an empty set, the type S must be *uninhabited*.

3. **Predicates.** Given a set member $a : S$, a predicate Q will produce a proposition $Q\,a : *$; it follows that Q may be typed as $Q : S \to *$.

4. **Implication.** Given a proof of $A : *$ (that is, a term $x : A$), an implication f creates a proof of $B : *$ (i.e. $f x : B$); clearly, $f : A \to B$.

5. **Universal quantification.** Consider the universal quantification $\forall x \in S\ Q(x)$, where Q is a predicate. Given any set member $x : S$, proof of universal quantification R is able to give a proof of the proposition $Q\,x : *$, which necessarily means that $R : \Pi x : S.\ Q\,x$.

For an example, consider the following proposition in first-order logic:

$$(\forall x, y \in S\ Q(x, y)) \implies (\forall u \in S\ Q(u, u))$$

Intuitively, the proposition holds because if $Q(x, y)$ is true for *any* x and y, then it definitely holds when x and y happen to be identical. The proposition above may be encoded in λP as the function type

$$(\Pi x : S.\ \Pi y : S.\ Q\,x\,y) \to (\Pi u : S.\ Q\,u\,u)\,.$$

A *proof* of the said proposition would be constituted by a λP *term* which has this function type. Since the proposition holds, the type is inhabited and we are indeed able to find such term:

$$\lambda z : (\Pi x : S.\ \Pi y : S.\ Q\,x\,y)\,.\ \lambda u : S.\ z\,u\,u$$

Observe how the proof term fully encodes the proof: it simply passes the arbitrary element $u \in S$ to the proof of $\forall x, y \in S\ Q(x, y)$, instantiating $Q(u, u)$. Since u was arbitrary, we

get $\forall u \in S\ Q(u, u)$.

2.2.6 Calculus of Constructions

The three extensions introduced by System F, System Fω and λP, namely *terms depending on types*, *types depending on types* and *types depending on terms*, respectively, can be combined into a single, "ultimate" typed λ-calculus system, named λC (a.k.a. *Calculus of Constructions*). Calculus of Constructions retains all the useful properties of $\lambda^{\rightarrow}$ like *uniqueness of types* or *strong normalization*; even better, well-typedness and type-checking are decidable, as proven by van Benthem Jutting in 1993 [22]. Term finding remains undecidable, but that much is understandable; we do not expect a computer to be able to prove an arbitrary proposition.

There are still ways to extend λC further. One could for example incorporate inductive types and the induction proof principle; this extension is called *Calculus of Inductive Constructions*. Another possibility of extending λC is taking the idea of types–kinds–sorts one step further and introducing an *infinite, cumulative hierarchy of universes*, in which $\Box_i \subseteq \Box_{i+1}$ for all i; such extension has been named the *Extended Calculus of Constructions*. Perhaps most importantly, both of these ideas (and others, too) form the theoretical basis for the interactive theorem prover Coq [7, 6].

CHAPTER 3
FORMALLY VERIFIED SOFTWARE

In this chapter, we survey formally verified software from four broad categories: application software, operating systems, distributed systems and file systems. The list is by no means exhaustive; its purpose is to showcase different practical approaches to verification of software taken by various authors or groups. To that end, brief description of the used tools and achieved results is included with each project.

We choose four formally verified file systems for fuzzing experiments: FSCQ, Yxv6 / Yggdrasil, Flashix and AtomFS. For this reason, extra space is dedicated to discussing more in-depth technical aspects of each.

3.1 Application software

CompCert (2006–2009) is a proven-correct C compiler; its formally verified front-end was introduced by S. Blazy et al [23], while the back-end was presented by X. Leroy [1]. CompCert parses a program written in a subset of the C language called Clight and gradually compiles it down to PowerPC assembly, passing through eight (!) intermediate languages in the process. Each language has precisely defined semantics, therefore by proving that the translation between two consecutive languages preserves semantics of the program, end-to-end (Clight to PPC) refinement proof can be obtained via composition. All CompCert code and proofs are written in Coq.

CakeML (2014–2019) is a verified implementation of a substantial subset of Standard ML, introduced by Kumar et al [24]; however, it is being continuously developed to this day [25, 26]. CakeML has been used to develop several verified applications, such as Unix-like tools grep, sort or cat, or a certificate checker for floating-point error bounds. Notably, the CakeML compiler itself has been bootstrapped as well [27]. For proofs and verification,

the CakeML project relies on an interactive theorem prover HOL4 (HOL stands for Higher Order Logic).

Ironclad Apps (2014), presented by Hawblitzel et al [28], are four apps running atop of OS, libraries and drivers, all verified end-to-end using Hoare logic. Hawblitzel et al write both the specification and implementation in Dafny and compile it to a verifiable assembly language called BoogieX86. The entire system can then be verified using the Boogie verifier, which relies on an automated SMT solver (Z3). The proof covers the actual assembly code that gets executed, and demonstrates indistinguishability from the app's high-level abstract state machine.

Vale (2017) is a language for expressing and verifying high-performance assembly code, developed at Microsoft Research with contributors from several universities [29, 30]. Vale allows the developer write assembly code annotated annotated with types, pre-/post-conditions and other features designed to help write correct code. The Vale tool parses the assembly code as well as the annotations and lifts them into Dafny, another imperative language targeting the .NET platform that is amenable to formal verification using SMT solvers, namely Z3. All relevant proof obligations are automatically discharged by Z3 before extracting executable assembly code for the target architecture. Vale has been used for developing verified cryptographic code, among other things.

CSPEC (2018) is a framework for formally verifying concurrent software, introduced by Chajed from MIT CSAIL [31]. Chajed notes that the key challenge in verifying concurrent software is the number of interleaved executions the developer has to consider; CSPEC framework reduces the number of variants by exploiting left- or right-commutativity of certain operations with respect to competing concurrent operations. The paper evaluates CSPEC by implementing CMAIL, a simple concurrent mail server. Similarly to Chajed's previous work on CIO-FSCQ [32], the development and proofs are done in Coq, and the runnable Haskell code is obtained by Coq's code extraction facilities.

3.2 Operating Systems

seL4 (a.k.a. *L4.verified*; 2009) is a verified implementation of a microkernel from the L4 family, developed by G. Klein et al [33, 2]. It constitutes one of the most successful applications of formal methods to realistic software. The final proof covers refinement of an abstract specification modeled in Isabelle/HOL by the compiled executable code. The project is also noteworthy for its development workflow: the design of the microkernel underwent several iterations in which the prototypes were written in Haskell; however, the final design was eventually manually rewritten and verified in C.

CertiKOS (2011–2019) is a framework for building certified OS kernels, developed at Yale [34, 35, 36, 37, 38]. CertiKOS was originally introduced in 2011, but its development has not ceased since. Initially introduced as a certified hypervisor for cloud environments with guarantees about correctness and absence of information leakage, CertiKOS has evolved into a case study of building a verified concurrent OS kernel with fine-grained locking, achieved by using "certified abstraction layers." These layers compose and enable refinement proofs while restricting interactions between different modules.

Hyperkernel (2017) is a verified OS kernel developed by the UNSAT group at University of Washington [39, 40]. Hyperkernel is in some aspects reminiscent of Yggdrasil [41] (described later), a verified file system also developed at University of Washington: specifications are written in Python as state machines with abstract state, all operations are required to be finite (i.e. all possible traces of an operation have finite and bounded length), and the verification work is off-loaded to an SMT solver (Z3). Contrary to Yggdrasil, however, the implementation of Hyperkernel is written in C, and verified against the specification at LLVM IR level.

Komodo (2017) is a formally verified reference monitor for ARM TrustZone, developed at Microsoft Research with contributors from Cornell University and Carnegie Mellon University [42]. Komodo implements features similar to those offered by Intel SGX, except it

does so in software: a privileged software monitor written in verified assembly code runs on the TrustZone platform and presents an API for managing secure user enclaves. An advantage of Komodo (compared to Intel SGX) is that its implementation is not tied to any firmware microcode or even hardware, enabling easier deployment of upgrades and security fixes. Komodo is built using Vale (described earlier), which depends on Dafny and Z3 under the hood for the verification work.

NiStar (2018) is an experimental kernel by Sigurbjarnarson et al, developed using Nickel, a framework tailored for verification of information flow [43]. The novel aspect of Sigurbjarnarson's work is the focus on flow of information between threads and processes and, in particular, the non-interference property. Interestingly, Sigurbjarnarson uses Coq to build and verify a non-interference metatheory to serve as a basis for Nickel; Nickel itself, however, uses SMT reasoning and Z3 to prove (or disprove) the necessary properties about concrete programs. NiStar builds on HiStar, a kernel introduced by Zeldovich et al (Stanford University) at 2011 [44].

Serval (2019) is a framework for developing automated systems software verifiers [45], authored by the UNSAT group from University of Washington. Serval includes many ideas from their previous work (Yggdrasil [41], Hyperkernel [39], NiStar/Nickel [43]), but extends them into a more general setting. Specifically, Serval lets developers write specifications in Rosette (an extension of the Racket language suitable for verification) and provides tools for lifting the implementation code into symbolic values (via symbolic evaluation), both of which are passed to an SMT solver (Z3) for automated discharge of proof obligations. Serval has been used as an alternative formal verification system for CertiKOS and Komodo, among others.

3.3 Distributed systems

Verdi (2015) is a framework for formal verification of distributed systems by J. R. Wilcox et al from University of Washington [46]. Verdi provides formalizations of various network

models with different fault modes; developers can choose which model is most suitable for their application. Moreover, Verdi allows for development of a correctness proof under an idealized error model first, then later effortlessly transfer that proof to a more realistic fault model. Verdi is implemented in Coq; to obtain executable code, users of Verdi are encouraged to extract an OCaml program from Coq and link it with a Verdi shim, which implements Verdi's network primitives. Verdi has been successfully used to develop a verified implementation of the Raft consensus protocol.

IronFleet (2015) is a methodology for proving safety and liveness properties of distributed systems, developed at Microsoft Research and presented in 2015 by Hawblitzel et al [47]. IronFleet's novelty lies in unique combination of TLA-style state machine refinement and verification based on Hoare logic: TLA state machines are used to reason about concurrency at the protocol level while ignoring complexities of the implementation, then Hoare logic is used to verify the implementation while ignoring concurrency. In terms of tooling, IronFleet depends on Dafny and Z3 for automated proof search; the final executable is obtained by translating Dafny to C# and compiling using a .NET compiler. Hawblitzel demonstrates use of the IronFleet methodology on development of two non-trivial distributed applications, a Paxos-based state machine replication library and lease-based sharded key-value store.

Chapar (2016) is a framework for verification of causally-consistent distributed key-value stores by Lesani et al from MIT [48]. Lesani describes a common abstract interface for key-value stores, together with operational semantics for implementations of this interface; a concrete implementation is said to be causally consistent if it refines the operational semantics. In addition, a novel proof technique called *well-reception* is presented, which is shown to be a sufficient condition for causal consistency and reduces the proof requirements to a small set of specific proof obligations for each implementation. The techniques are demonstrated on several examples of practical key-value stores. Chapar is formalized in Coq; runnable code for the examples is obtained by extracting OCaml code from the

Coq definitions.

3.4 File systems

COGENT (2016) is a purely functional language developed by contributors from CSIRO and UNSW, Australia [49]. The COGENT compiler is able to generate three artifacts from COGENT source: C code with the same behavior, a formal specification of the behavior in Isabelle/HOL, and a valid proof that the C code refines the formal specification. This may not appear very useful at first sight, as the COGENT code may still contain bugs; however, a developer may supply an even more abstract specification of the desired behavior, and prove that it is refined by the COGENT-generated specification. By transitivity of refinement, such construction proves correctness of the COGENT-generated C code. COGENT has been used to implement and verify a number of file systems: BilbyFS, ext2fs, vfat and f2fs.

3.4.1 FSCQ

FSCQ and its derivatives (2015–2018) were developed at the Computer Science & Artificial Intelligence Lab at Massachusetts Institute of Technology by H. Chen, T. Chajed, A. Ileri and others [50, 51, 52, 32, 53].

Crash Hoare Logic

Crash Hoare Logic (CHL), introduced by H. Chen et al in 2015 [50, 51], is an extension of Hoare logic for the purposes of proving crash-consistency in file systems.

When Hoare logic is used to describe semantics of file system operations, every operation is associated with two conditions: a pre-condition and a post-condition. Both describe state of the file system; the former is a state predicate which holds prior to invoking the operation, while the latter is a state predicate that holds after the operation completes. Note that this directly corresponds to what the proof of correctness needs to show, namely that the operation establishes the post-condition upon completion, assuming the pre-condition

holds at the start of the operation.

However, this approach alone is not sufficient for reasoning about crashes and crash consistency of the file system. A crash may occur at any point during execution of the operation, even before the post-condition is established. Additionally, file systems need recovery routines to roll back (or forward) into a consistent state; it is not immediately obvious how recovery can be incorporated into pre- or post-conditions alone. To solve this problem, Chen et al propose extending Hoare logic with *crash-conditions*, *logical address spaces*, and *recovery execution semantics*. All three will be described in the following paragraphs.

The *crash-condition* is another state predicate associated with an operation, just like pre- and post-condition. If, during execution of the operation at hand, a crash occurs, the crash-condition predicate describes the set of all possible states the file system may be in right before the crash. The disk model of CHL specifies that after the system is restarted, the disk will have non-deterministically picked one of these possible states. This can be contrasted with a model where the disk maintains only a single state throughout the execution: albeit simpler, this model cannot capture asynchronous writes, and therefore forces the disk to operate in a synchronous mode, which comes with a considerable performance penalty in practice.

Another concept that CHL introduces is *logical address spaces*. Chen observes that many concepts in file systems are examples of logical address spaces: the disk is an address space from disk-block numbers to disk blocks, inode layer is an address space from inode numbers to inode structures etc. He also notes that specifying the disk state predicates directly, everywhere throughout the file system stack, is rather cumbersome and error-prone. The concept of logical address spaces is illustrated by an example: the first abstraction one typically wishes to build on top of a persistent disk is a persistent write-ahead log with atomic transactions. At this level of abstraction, the disk may be safely viewed is a simple map from block numbers to blocks even in presence of multiple in-

flight writes; the log recovery routine guarantees that all committed transactions will be (eventually) replayed. Therefore, we wish to avoid directly specifying the set of all possible disk states in pre-/post-/crash-conditions of the log operations. Instead, the conditions specify the (synchronous-like) logical address space presented by the persistent log, and a *representation invariant*: a disk state predicate that describes the set of all possible disk states such that the log presents the given logical address space.

Finally, *recovery execution semantics* allows to "erase" the crash-condition from the public interface of the file system. As described above, crash-condition of an operation describes the set of all possible disk states during execution of the operation, right before the crash. However, when a crash occurs, the recovery routine makes sure the file system remains consistent on restart. In that sense, the operation combined with the recovery routine has no crash-condition, and the post-condition describes the state of the file system either after the operation completes successfully without crashing, or after the recovery finishes successfully (with possibly many additional crashes during recovery).

Architecture

High-level overview of FSCQ is given in Figure 3.1. Both Crash Hoare Logic and FSCQ are Coq code; Coq is the interactive theorem prover used for development of FSCQ. Coq fulfills two important roles for FSCQ: first, it verifies all the proofs of correctness relating pre-conditions, post-conditions and crash-conditions, and second, it extracts executable Haskell code from the Coq definitions. The Haskell code is then compiled by GHC (Glasgow Haskell Compiler) and linked with Haskell FUSE driver and libraries to produce a FUSE file system server implementing FSCQ. When the FUSE server runs, it registers itself into the VFS (Virtual File System) layer in Linux kernel, and awaits upcalls from the kernel. When a user wishes to perform a file system operation (such as renaming a file), a system call (in this case, `rename`) is issued to the Linux kernel. The kernel redirects the call towards the FUSE server (running in user-space), who is actually responsible

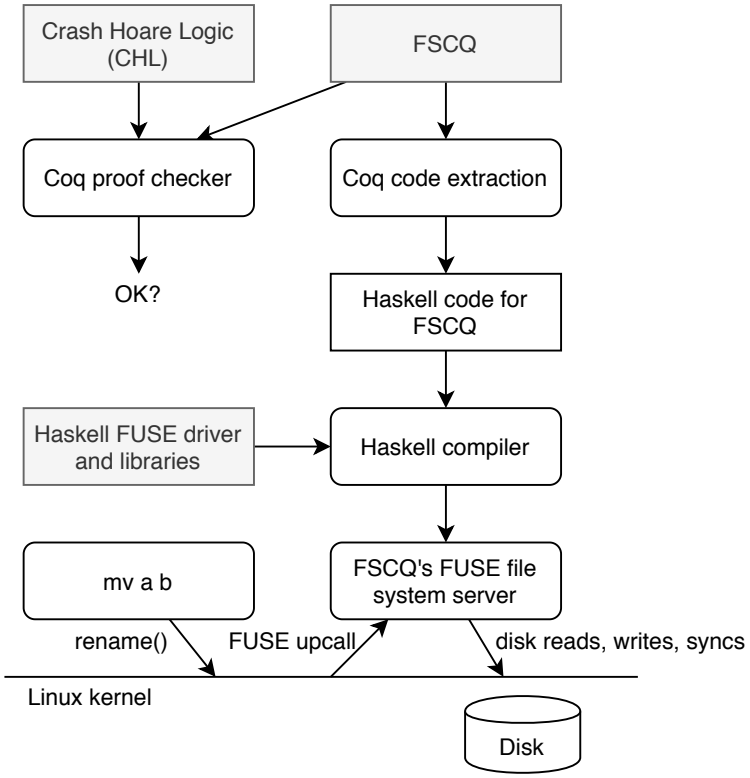

Figure 3.1: Overview of FSCQ, adapted from [51]. Grey boxes indicate hand-written code. Rectangle boxes show source code, rounded boxes are processes.

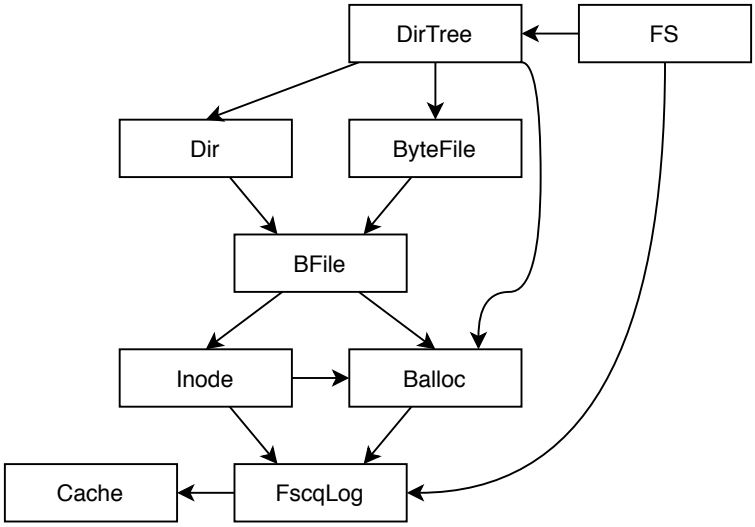

Figure 3.2: High-level architecture of FSCQ, originally presented in [51].

for correctly handling the system call, issuing disk reads, writes, and other operations as necessary.

Figure 3.2 displays how FSCQ code is modularized. The core modules are `Cache`, which provides buffer cache, and `FscqLog`, which implements a persistent log with atomic transactions (as briefly described in the previous section). Modules `Balloc` and `Inode` are the only direct users of the persistent log; the former is used for bitmap-based allocation of blocks, while the latter implements the inode layer. The `BNode` module uses both `Balloc` and `Inode` to provide a block-based file abstraction, i.e. files which are a list of blocks. Both directories (module `Dir`) and the byte-based file abstraction (module `ByteFile`) are implemented on top of the block-based files. Finally, the module `DirTree` combines the directories and byte-based files into a coherent directory tree structure, and the module `FS` implements all required file system operations using transactions.

Deferred writes and metadata-prefix

A follow-up paper, also by H. Chen et al, was released later in 2017 [52], describing D-FSCQ (Deferred-write FSCQ), a variant of FSCQ with verified implementation of common performance-oriented features like deferred writes, group commit, direct writes for file data

etc.

In order to improve performance, two optimizations are commonly implemented in file systems: deferred/batched writes, and log-bypass writes. Deferring writes lets the file system buffer pending writes in a volatile memory, which allows the writes to be sent to the disk in larger batches, improving throughput. In case of a crash, however, all buffered writes are lost and never recovered, because they never reached the persistent log. Log-bypass writes are useful in cases where data blocks of an existing file are being updated: instead of writing the data block into the log first, and then updating data block of the actual file, the write is never logged, which often offers substantial speedups. The drawback is that the writes may be reordered by the disk controller, and consequently appear out-of-order with respect to each other and with respect to metadata updates.

To counteract these potentially destructive side-effects, two additional operations are supported by file systems: `fsync` and `fdatasync`. `fdatasync` ensures that upon its successful completion, all data block writes to a given file have been persisted to the disk. `fsync`, in addition to providing the same guarantee, also guarantees persistence of all metadata writes related to the same file. Unfortunately, proper use of these two operations is far from intuitive. Consider a simple task of atomically rewriting a given file with new content; an application needs to make sure that either the original file is left intact, or it has been completely rewritten with new data. Correct solution to this problem involves creating a temporary file with the new data, issuing `fdatasync` on the temporary file, calling `rename` to replace the original file and issuing `fsync` on the *containing directory*. This approach is not immediately obvious and therefore error-prone.

DFSCQ formalizes these concepts into the *metadata-prefix specification*: the semantics of `fdatasync` is identical to what has been described above, but `fsync` provides a stronger guarantee, namely that all metadata writes (to any file, not just the given one) have reached persistent storage. Additionally, the file system is allowed to flush either data block writes in any order, or metadata writes in order up to some point in time; this en-

sures that while data block writes may be arbitrarily reordered, the metadata writes always correspond to some prefix of operations performed on the file system since the last `fsync`.

The metadata-prefix specification is then projected into the corresponding pre-, post-, and crash-conditions of the file system operations. Since the metadata effectively correspond to a directory tree structure, the DFSCQ condition predicates describe a sequence of trees since the last `fsync`. When a crash occurs, the file system may recover to an arbitrary tree in this sequence; notably, it will never recover to a tree not included in this sequence, because the metadata writes are never reordered. That being said, the same is not true of log-bypass data writes; the DFSCQ predicates reflect this by rewriting the data block in all trees in the sequence.

3.4.2 Yxv6 / Yggdrasil

Yggdrasil (2016) is a toolkit targeted at helping programmers write formally verified file systems, introduced by Sigurbjarnarson et al at the University of Washington. [41] Sigurbjarnarson also demonstrates the use of Yggdrasil by implementing a verified file system Yxv6, a spin-off of Xv6's file system; Xv6 is a simple Unix-like teaching operating system.

Yggdrasil

The Yggdrasil toolkit views both specification of the file system and its implementation as a persistent state machine. Concretely, let s denote an arbitrary state, $\boldsymbol{x}$ an external input (e.g. data to write), $\boldsymbol{b}$ a crash schedule, and f an operation transition function, then $f(s, \boldsymbol{x}, \boldsymbol{b})$ denotes the next state of the system after the operation associated with f. Additionally, Yggdrasil maintains the notion of a *consistency* or *well-formedness* state predicate, $\mathcal{I}(s)$, which determines if s is a consistent state, and a *state equivalence relation* $s_1 \sim s_2$, allowing to declare equivalence between states of two related systems (state machines). For conciseness, the last two concepts are contracted into definition of a *consistent-state equivalence relation* $s_1 \sim_{\mathcal{I}_1,\mathcal{I}_2} s_2$: if s_1, s_2 are arbitrary states of two state machines, $\mathcal{I}_1, \mathcal{I}_2$

consistency predicates on their respective state spaces, and $\sim$ an equivalence relation between their state spaces, then

$$s_1 \sim_{\mathcal{I}_1,\mathcal{I}_2} s_2 \quad \stackrel{\text{def.}}{\iff} \quad \mathcal{I}_1(s_1) \wedge \mathcal{I}_2(s_2) \wedge s_1 \sim s_2.$$

User of the Yggdrasil toolkit needs to build two state machines: one corresponding to the specification, and one corresponding to the implementation. Both consist of three components: representation of a state, definitions of the operations, and the consistency invariant. Moreover, a state equivalence relation must be defined, relating equivalent states of the two machines. The key difference between the two state machines is that the specification is allowed to use abstract data types and abstract data structures; on the other hand, the implementation should be fully executable.

Given the two state machines and the equivalence relation, Yggdrasil toolkit attempts to prove several theorems about the state machines, with the ultimate goal of proving that the implementation state machine is a *crash-refinement* of the specification state machine. The theorems build on one another and are as follows:

1. **Crash-free equivalence.** Let f_1, f_2 be operation transition functions, one from either machine, and $\mathcal{I}_1, \mathcal{I}_2$ their consistency predicates, then f_1 and f_2 are *crash-free equivalent* if

   $$\forall s_1, s_2, \boldsymbol{x}.\ s_1 \sim_{\mathcal{I}_1,\mathcal{I}_2} s_2 \implies f_1(s_1, \boldsymbol{x}, \boldsymbol{true}) \sim_{\mathcal{I}_1,\mathcal{I}_2} f_2(s_2, \boldsymbol{x}, \boldsymbol{true}),$$

   where $\boldsymbol{true}$ denotes a crash schedule with no crashes. Intuitively, crash-free equivalence means that if the two machines start in two arbitrary but equivalent states and both perform the crash-free equivalent operation, they will again end up in two equivalent states (assuming no crashes occur).

2. **Recovery idempotence.** Let r be a recovery transition function, i.e. a special tran-

sition function which takes no external input. We say that the recovery function r is idempotent if

$$\forall s, \boldsymbol{b}.\ r(s, \boldsymbol{true}) = r(r(s, \boldsymbol{b}), \boldsymbol{true}).$$

Intuitively, the recovery function is idempotent if the machine recovers to the same state no matter how many crashes occur during recovery.

3. **Crash refinement with recovery.** Let f_1, f_2 be operation transition functions, one from either machine, $\mathcal{I}_1, \mathcal{I}_2$ their consistency predicates, and r a recovery transition function, then f_2 with r is a crash refinement of f_1 if f_2 is crash-free equivalent to f_1, r is idempotent and

$$\forall s_1, s_2, \boldsymbol{x}, \boldsymbol{b}_2.\ \exists \boldsymbol{b}_1.\ s_1 \sim_{\mathcal{I}_1,\mathcal{I}_2} s_2 \implies f_1(s_1, \boldsymbol{x}, \boldsymbol{b}_1) \sim_{\mathcal{I}_1,\mathcal{I}_2} r(f_2(s_2, \boldsymbol{x}, \boldsymbol{b}_2), \boldsymbol{true}).$$

Intuitively, crash refinement with recovery says that every state produced by f_2, after recovery has successfully finished, must be equivalent to some state produced by f_1, no matter what crashes occur throughout.

4. **No-op.** A transition function f with a recovery function r is a *no-op* if r is idempotent and

$$\forall s_1, s_2, \boldsymbol{x}, \boldsymbol{b}.\ s_1 \sim_{\mathcal{I}_1,\mathcal{I}_2} s_2 \implies s_1 \sim_{\mathcal{I}_1,\mathcal{I}_2} r(f(s_2, \boldsymbol{x}, \boldsymbol{b}), \boldsymbol{true}).$$

No-ops capture the notion that the state machine may be running operations in the background (e.g. inode garbage collection) which do not affect the externally-exposed state.

5. **System crash refinement.** Given two state machines F_1, F_2 and a recovery function r from F_2, F_2 is a *crash-refinement* of F_1 if every operation transition function in F_2 with r is either a crash-refinement of the corresponding transition function in F_1, or a no-op.

Yggdrasil is targeted at verifying Python code, so both state machines are Python classes: state is represented by instance variables, and allowed operations, consistency invariant and the equivalence relation are instance methods. To actually prove the theorems, Yggdrasil relies on an SMT (satisfiability modulo theory) solver, namely Z3. In order to build an SMT-compatible encoding of the transition functions, Yggdrasil performs symbolic evaluation of the instance methods to recover expressions for updating the state. This places some restrictions on what the transition function may do, the most important of which is loop length and recursion depth. Yggdrasil unrolls loops and expands recursive functions without bounding depth, so it *will* fail on non-finite code; the user of Yggdrasil must make sure that e.g. all loops are properly bounded.

Once the transition functions are expressed in a form suitable for the solver, the theorems above are proved by attempting to find a model which satisfies their negation, i.e. a counterexample. An advantage of this approach is that if the theorem does *not* hold, the counterexample, which can often guide the programmer towards the offending bug, is readily available. Yggdrasil explicitly supports this approach by providing a counterexample visualizer, which will produce a human-readable trace of the file system operations leading up to the point where the theorem is violated.

Yggdrasil also supports transition function optimization via rewriting the abstract syntax tree recovered by symbolic execution. User-supplied optimizers need not be proven correct; Yggdrasil automatically re-verifies the optimized version to validate the optimization. A simple disk flush optimizer is supplied with Yggdrasil: the optimizer greedily removes every disk flush from the transition function unless the removal causes violation the crash refinement theorems. After the verification-optimization process is finished, the resulting Python code is compiled using Cython and linked with FUSE into native Python modules.

Figure 3.3 visualizes the development flow of a file system built using Yggdrasil. The shaded boxes represent trusted components of the system; their correctness is not verified.

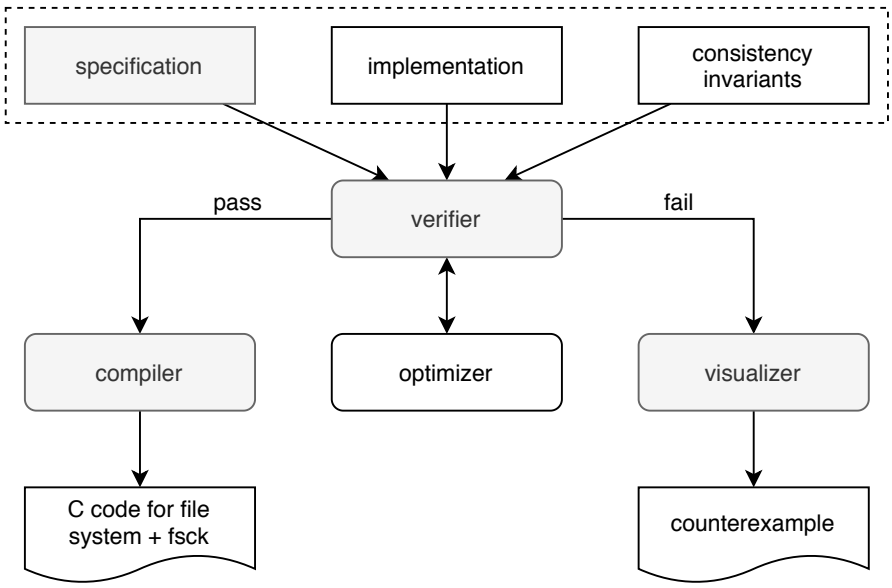

Figure 3.3: Diagram of Yggdrasil development flow, taken from [41].

The dashed region at the top marks components which a file system programmer must supply: high-level specification of the file system, the implementation that is to be proved to be a crash-refinement of the specification, and the necessary consistency invariants. The rounded boxes are all parts of Yggdrasil, and the curved boxes at the bottom are outputs of the Yggdrasil toolkit.

Yxv6

Yxv6 file system actually uses five Yggdrasil-verified layers stacked on top of each other, as shown in Figure 3.4. The reasoning behind this step is both conceptual and practical: properly modularizing the code supports human understanding and ease of maintainability, while simultaneously making the associated verification problems tractable. Indeed, Sigurbjarnarson notes that verifying the bulk of Yxv6's implementation code against its specification directly (with no intermediate layers) is an intractable problem even for today's state-of-the-art SMT solvers.

The shaded, rounded boxes in Figure 3.4 represent abstract specifications, whereas rectangle boxes are the actual implementations. Observe that at the bottom of the figure, a

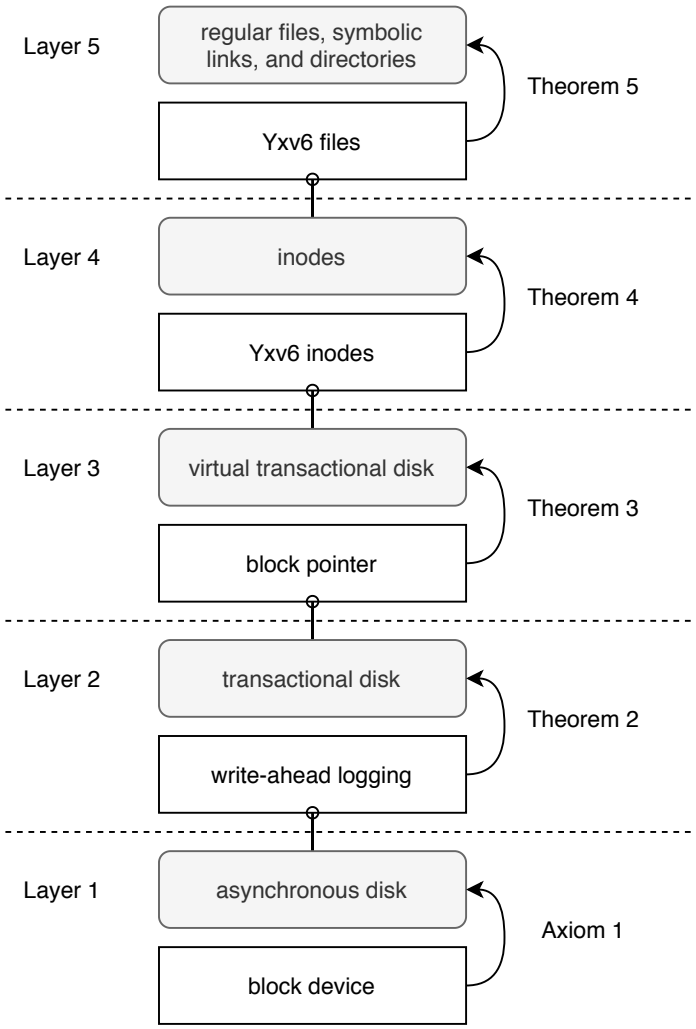

Figure 3.4: The stack of verified layers that make up Yxv6. This illustration was originally presented in [41].

block device implements the asynchronous disk abstraction, but this must be assumed as an axiom. To implement the transactional disk abstraction, a write-ahead persistent log with atomic transactions is built on top of the asynchronous disk abstraction. In a similar manner, the virtual transactional disk abstraction (which differentiates between virtual and physical disk addresses) is built using regular transaction disks, and the inodes layer abstraction is implemented using virtual transactional disks. Finally, the POSIX file system specification is implemented in layer 5, relying on the inodes abstraction.

3.4.3 Flashix

Flashix (2012–2017) is a verified file system for flash memory, developed by G. Ernst et al [54, 55, 56, 57, 58, 59] at University of Augsburg. The formal framework used in Ernst's work is *Abstract State Machines* (ASMs), introduced in the 1990s by Gurevich [60] under the name *evolving algebras*. An overview of the framework is given e.g. by Börger [61]; summary of the key concepts in ASMs (as related to Flashix) is given in the following subsection. Flashix is built with KIV, an interactive theorem prover actively developed at the University of Augsburg. KIV features integration into Eclipse, a well-known IDE for Java-based platforms; the publicly available code for Flashix is in Scala, a functional language targeting the Java Virtual Machine.

Abstract State Machines

Abstract state machines may be considered an extension of regular finite-state machines. An *abstract state machine* M is a triplet

$$M = (\mathbf{x} : St, \text{Init}, p)$$

where $\mathbf{x} : St$ stands for the vector of program variables (i.e. the state space of M), Init $\subseteq St$ is a set (or a predicate) characterizing the initial states, and p is the main program describ-

ing the computational steps of M. Computation of an abstract state machine starts in one of the states from Init and proceeds by executing the program p step by step; the intermediate states are recorded and form a *run* (also called an *interval*). The program structure is not described here for brevity, but syntactically resembles a regular programming language with constructs like assignment, sequential composition, iteration, or procedure call; perhaps more interesting is the inclusion of a non-deterministic operator of choice, which may be used to specify non-deterministic machines.

A *data type like abstract machine* M is an abstract state machine

$$M = \left(\mathbf{x} : St, \text{Init}, \left\{\text{Op}_j\right\}_{j \in \mathcal{J}}\right)$$

where each operation $\text{Op}_j, j \in \mathcal{J}$ is a quadruplet $\left(\text{pre}_j, in_j, p_j, out_j\right)$: the program p_j gives the computational steps for the operation, reading input from formal input parameters in_j and writing output to formal output parameters out_j, provided the precondition pre_j holds. Intuitively, a data type like abstract machine is equivalent to an abstract data type (e.g. a dictionary or a priority queue): the machine state vector $\mathbf{x}$ captures the internal state of the data structure, while operations Op_j are provided to the user for manipulation with the data structure.

To achieve modularity, machines may be composed using *submachine composition*: given an abstract state machine X,

$$X = \left(\mathbf{xx} : St^X, \text{Init}^X, \left\{\text{Op}_j^X\right\}_{j \in \mathcal{J}}\right)$$

we can get a composed machine $M(X)$,

$$M(X) = \left((\mathbf{mx}, \mathbf{xx}) : \left(St^M, St^X\right), \text{Init}^M \cap \text{Init}^X, \left\{\text{Op}_k^M\right\}_{k \in \mathcal{K}}\right)$$

where

$$M(\cdot) = \left(\mathbf{mx} : St^M, \text{Init}^M, \left\{\text{Op}_k^M\right\}_{k\in\mathcal{K}}\right)$$

is called a *supermachine with a hole*. In addition to the usual syntactical elements, the programs of $M(\cdot)$ may also include operations Op_j^X; the use of X when referring to the submachine operations corresponds to the interface contract between $M(\cdot)$ and X.

At this point, the last major missing ingredient is the definition of machine refinement. Ernst notes that a meaningful refinement relation should be *reflexive* and *transitive*; additionally, refinement should aim to preserve the externally visible behavior while hiding the internal state, i.e. if C refines A, we would like to use A to reason about the expected behavior of the system, even though the actual code might be provided by C. This idea is called *substitutivity* and is the central goal of any refinement proof. Refinement of abstract state machines is defined in terms of their runs: a data type like ASM C refines a data type like ASM A if for every run of C invoking some operations $\left\{\text{Op}_j^C\right\}_{j\in\mathbf{j}}$ and observing inputs $\mathbf{i}$ and outputs $\mathbf{o}$, there exists an equally-long run of A invoking the same operations $\left\{\text{Op}_j^A\right\}_{j\in\mathbf{j}}$ and observing identical inputs $\mathbf{i}$ and outputs $\mathbf{o}$. The machine C is allowed to diverge, but only if A also diverges.

However, working with traces directly when proving refinement may be a bit unwieldy. Therefore, Ernst proceeds to define a special case of refinement when the two machines A and C remain in lock-step all throughout the execution: the *forward simulation of ASMs*. Concretely, machine C refines the machine A if there is a *forward simulation condition* $R(\mathbf{ax}, \mathbf{cx})$ relating the states $\mathbf{ax}$ of A and the states $\mathbf{cx}$ during an arbitrary run; of course, the criterion R must fulfill several common-sense criteria, which we omit for brevity.

Treatment of crashes

The basic idea behind extending the ASM machinery to include crashes is to allow a non-deterministic transition into a "crashed" version of a regular state, which signifies that the system has not finished transitioning to the regular state. The crash transition is then

immediately followed by two other transitions: the first transition, described by a relation Cr, moves the system from the crashed state into some regular state, which is seen as realizing effects of the crash. The second transition is a recovery transition that presents an opportunity for the system to put itself into a consistent state after crash.

Ernst presents this design in a more general setting of *transitioning systems*; the definition for ASMs contracts the non-deterministic crash transition and the crash-effect realization transition into a single step. Concretely, a *data type like abstract state machine with crash behavior* is an ASM

$$M = \left(\mathbf{x} : St, \text{Init}, \left\{\text{Op}_j\right\}_{j \in \mathcal{J}}, \text{Cr}\right)$$

where Cr : $St \times St$ is a relation that describes the possible transitions caused by a crash. After a Cr-transition (i.e. a crash) a specially designated recovery operation $\text{Op}_{rec}, rec \in \mathcal{J}$ is implicitly invoked, allowing the state machine M to perform the appropriate recovery steps. All the important theorems building on top of the definition of ASM (e.g. submachine composition, machine refinement, forward simulation etc.) are then extended and reproved in the crash-aware setting; since it is a largely technical undertaking, the reader is advised to consult the original source [59] for details.

Implementation of Flashix

Building directly on top of submachine composition and submachine refinement theorems, Flashix is composed from several layers of separately-verified implementations of well-defined abstractions (Figure 3.5). We shall describe the top half of the layer stack, since the bottom half is largely concerned with details of flash memory and not directly relevant to our work.

1. **POSIX.** The most abstract specification Flashix is ultimately targeting is the POSIX file system specification. Flashix models state of a POSIX-compliant file system as

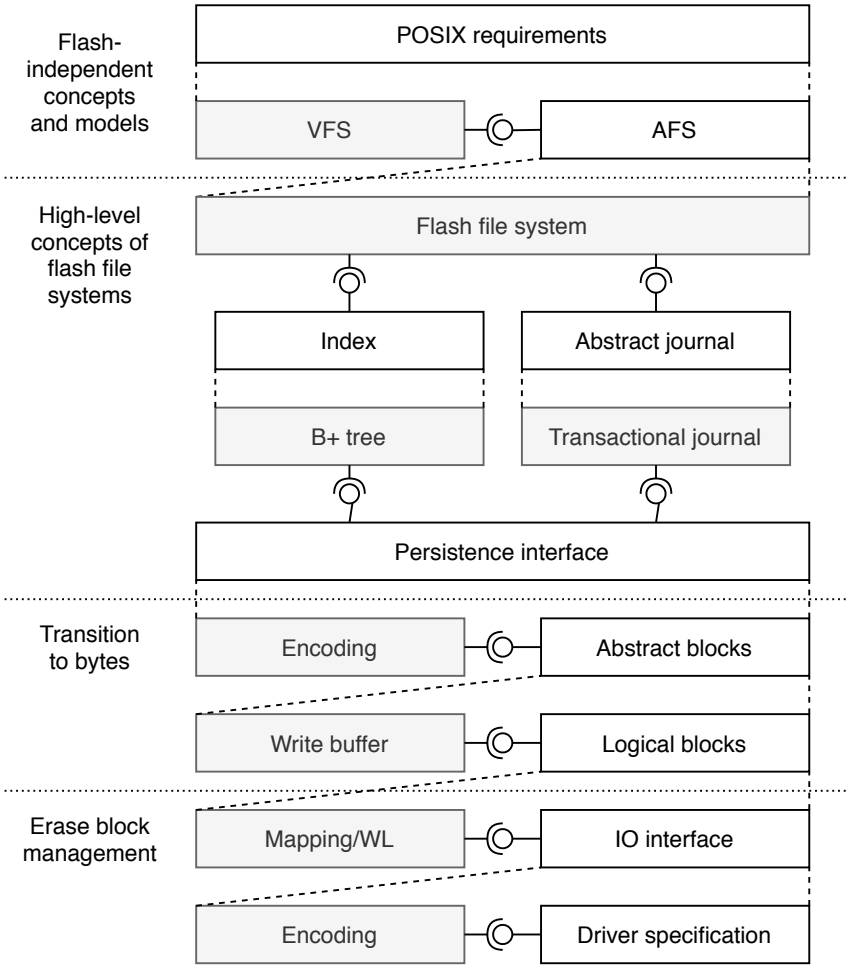

Figure 3.5: Overview of Flashix architecture, taken from [59] with slight modifications. White boxes are specifications, while shaded boxes represent implementations. Dashed lines between a specification and an implementation indicate the implementation refines the specification. The lollipop notation (familiar from UML) is used to denote that an implementation is internally a consumer of another abstraction.

a recursively defined *file system tree*, a *file store* mapping valid file identifiers to file contents, and a *registry of open file handles*. A file handle (direct equivalent of a file descriptor) consists of a file identifier, an offset within the file, and a mode restricting permitted operations on the file. Operations of a POSIX-compliant file system are realized by algebraic operations on the file system tree. Ernst formally proves several common-sense invariants that define consistency of the POSIX model, such as the root is a directory, or the file identifiers mapped in the store are precisely the union of file identifiers in the tree and in the registry of open handles. Notably, Ernst observes that even at this level of abstraction, atomicity alone is not sufficient to reason about crash-safety: since POSIX explicitly allows opened files to be deleted from the file system tree, the file system has to manage orphaned nodes, which in the case of a crash necessitates a recovery routine.

2. **Virtual File System (VFS).** Virtual File System directly refines the POSIX specification into a concrete implementation, relying on AFS (Abstract File System) abstraction to provide more basic primitives. The file system tree from POSIX is broken down into two mappings from inode numbers to files and directories, respectively; a directory is represented as a mapping from children's names to their inodes, while a file is realized as a mapping from natural numbers to pages (with unmapped naturals representing the all-zero page, which is suitable for sparse files). Some proven invariants include zero is never used as a valid inode number, no inode number is mapped to both a file and a directory, etc.

3. **Flash file system.** Core of the flash file system coalesces the three kinds of objects exposed by AFS (inodes, directory entries, file pages) into a unifying concept of a "node" identified by a key. Additionally, an indirection is introduced when resolving a key into the corresponding node: the key is first looked up in a RAM index to find the address of the node, then the address is read from the transactional journal

to obtain contents of the node. The indirection is necessary to facilitate garbage collection: nodes may be relocated to a different address during garbage collection, but this is an implementation detail of a flash-based persistence solution and must not leak into upper layers.

4. **Transactional journal.** Purpose of the transactional journal is to provide atomic multi-node writes to the underlying flash device even in presence of crashes. At this level, (logical) blocks are represented as lists of nodes: address of a node is given by a logical block number and a natural number, specifying position of the node in the list. Moreover, every node is accompanied by a metadata header detailing information about grouping of the node in the logical block.

5. **Index implementation (B^+ tree).** The index provides an efficient mapping between node keys and node addresses; implementation is based on the well-known B^+ tree data structure. However, as Ernsts notes, the application of B^+ tree is complicated by the fact that the tree is never completely loaded into RAM, and during normal operation the tree is continually changing, but the changes are not immediately propagated to the underlying flash device (for performance reasons). The mappings are stored in index blocks, which are distinct from blocks used by the transactional journal.

3.4.4 AtomFS

AtomFS (2019) is a verified concurrent in-memory file system introduced by Zou et al [62] at Shanghai Jiao Tong University. Zou builds the concurrency proof using an original framework named Concurrent Relational Logic with Helpers (CRL-H), which is based on a novel combination of local rely-guarantee reasoning, relational specifications, and a so-called helper mechanism. CRL-H and all AtomFS proofs are written in Coq. As opposed to the previous file systems, AtomFS explicitly avoids considering crashes in its specifications. Instead, it focuses on another important aspect of file systems, namely atomicity

and behavior under workloads with heavy concurrency. Note that parallelism has been a significant omission in specifications of the file systems we have described so far.

Towards file system atomicity with helpers

In order to prove AtomFS safe with respect to concurrency, Zou et al choose to show that the implementation of the file system is a contextual refinement of an atomic specification. Contextual refinement is a relation between an implementation and a specification, defined using observable event traces: if every observable event trace producible by the implementation can also be somehow generated by invoking the corresponding specification, the implementation contextually refines the specification. The specification that AtomFS provably refines is an atomic specification, i.e. it assumes all statements of an operation execute atomically. Prior work has shown that contextually refining an atomic specification is in fact equivalent to linearizability, which is (intuitively) the notion that there exists an order of the concurrent operations such that executing the operations sequentially in this order would yield the same responses.

The most straightforward way to prove contextual refinement is to establish forward simulation relations, which directly relate abstract states of the specification and concrete states of the implementation; the goal is to show that the concrete implementation "simulates" the abstract specification. In greater detail, the specification consists of an abstract state (which represents the concrete state on a logical level) and abstract operations (which describe, logically, how the abstract state is affected by the operations); the contextual refinement proof then demonstrates an abstraction relation, an invariant that holds over pairs of abstract and concrete states.

Intuitively, a valid approach would be to investigate the concrete implementation of every operation and find a specific place in the code where the concrete state transitions from the initial state to a state where the effect of the operation has taken place. Such a place or point in code is called linearization point of the operation, for obvious reasons;

since the linearization point is located statically in the code and does not move, it is also sometimes called a fixed linearization point.

Unfortunately, Zou et al give a counterexample which clearly demonstrates that an effort to find fixed linearization points in order to verify a file system with fine-grained (i.e. per inode) locking is bound to be unsuccessful. Consider two racing operations, `rename` and `mkdir`, where the `mkdir` operation is creating a directory in the sub-tree which `rename` is renaming. It is possible that `mkdir` first descends into the sub-tree, but while it is busy creating the directory, the `rename` operation renames the entire tree and successfully finishes first, followed by the (also successful) `mkdir` operation. However, note that this is not a linearizable trace: during serial execution, the `mkdir` operation would necessarily fail, since the sub-tree were moved elsewhere by the `rename` operation. Zou calls this occurrence the path inter-dependency phenomenon. In order to get the correct trace, `rename` would have to "help" `mkdir` commit its effect first, causing the linearization point of `mkdir` to reside *outside* of `mkdir`'s code. Such linearization points are called external LPs. Luckily, `rename` is the only file system operation in the POSIX specification which may break path integrity of other operations, and consequently leads to external LPs.

"Helping" other threads is in itself a concept known primarily from lock-free and wait-free algorithms. If an algorithm is to be wait-free, some sort of load-balancing mechanism needs to be implemented so that less busy threads may take on work from overloaded threads instead of idly waiting. However, Zou notes that the techniques cannot be directly applied to file systems, because the concurrency in a file system is implicit rather than explicit as in a parallel algorithm: the supporting global information such as thread identifiers or thread's intended operations are missing. Additionally, the helping thread (a.k.a. "helper") may need to help an unbounded set of competing threads, and must consider the order of the helped operations, since some orders may yield incorrect results. Finally, the path inter-dependency can also be recursive: consider the case where one thread is renam-

ing a subtree A, another thread is renaming a subtree B located within A, and yet another thread is running `stat` on a file within B. When the first thread wants to help, it must help the `stat` operation finish first lest an unlinearizable trace is obtained.

To solve the problems outlined above, Zou et al introduce a *helper mechanism for file systems*, featuring ghost state and a linearize-before relation. Ghost state is a notion familiar from other work in formal specification; it has no reflection in the concrete implementation as it fulfills its role exclusively on the abstract side, facilitating the formal proof. In Zou's work, the ghost state supplies the context that is not explicitly given in file systems, namely identifiers of other threads and their intended operations. The linearize-before relation, on the other hand, is used by the helper to decide the helping set (i.e. which threads need to be helped) and the helping order (i.e. in what order the threads in the helping set must be helped). The ghost state and the linearize-before relation are combined in a single, abstract operation named `linothers`: `linothers` uses the ghost state to detect other threads, then uses the linearize-before relation to determine which to help and in what order, and finally helps them.

Just to re-iterate, keep in mind that all of the concepts described above exist only in proofs. The overarching goal is to prove that the concrete implementation contextually refines an abstract specification that is atomic; helping other threads is simply an idea about how to augment the abstract specification so that it can be refined by the concrete code. Actual implementation of AtomFS does not "help" other threads in any way, and the only synchronization is facilitated by per-inode locks and lock coupling (described later).

Concurrent Relational Logic with Helpers

In summary, Concurrent Relational Logic with Helpers (CRL-H) combines two pre-existing approaches to verification, local rely-guarantee reasoning and relational specifications, and extends them with the notion of helpers developed in the previous subsection.

Rely-guarantee reasoning uses rely- and guarantee-conditions (reminiscent of pre- and

post-conditions from Hoare Logic) to reason about concurrency in programs. Rely-condition describes how the state of the system may evolve on its own, without interaction with the thread at hand, presumably due to interactions with other threads. On the flip side, guarantee-condition constrains how the thread at hand may evolve the state of the system. Both rely- and guarantee-conditions are typically defined as a predicate on pairs of current and next system states. Local rely-guarantee reasoning introduces an invariant describing the boundary between local and shared context, and a frame rule, which allows to strip the unrelated context from the proof result and facilitates easier reuse of previously proven-correct concurrent routines.

Relational specification has been described earlier: it is the idea that the specification consists of abstract state and abstract operations, the implementation is made up of concrete state (C memory) and concrete implementation (C code), and in order to prove contextual refinement of the specification by the concrete implementation, we must establish an abstraction relation spelling out the precise relationship between abstract and shared states and the corresponding operations.

CRL-H throws another two concepts into the mix: helper metadata and a roll-back mechanism. Helper metadata is the ghost state introduced by CRL-H. Users of the CRL-H framework may define what the metadata is precisely; the metadata should contain all information necessary to determine the helping set and helping order, which may vary depending on the use-case. The roll-back mechanism solves an interesting problem: when a thread performs the (abstract) operation `linothers`, state of the abstract system may in fact suddenly jump several steps ahead of the concrete state, breaking down the forward simulation argument, i.e. the abstraction relation. The roll-back mechanism allows to undo the extra operations on the abstract state, helping to re-establish the correspondence between abstract and concrete states.

When the user of the CRL-H framework wishes to define a file system specification, they need to supply abstract operations over the file system abstraction, rely- and guarantee-

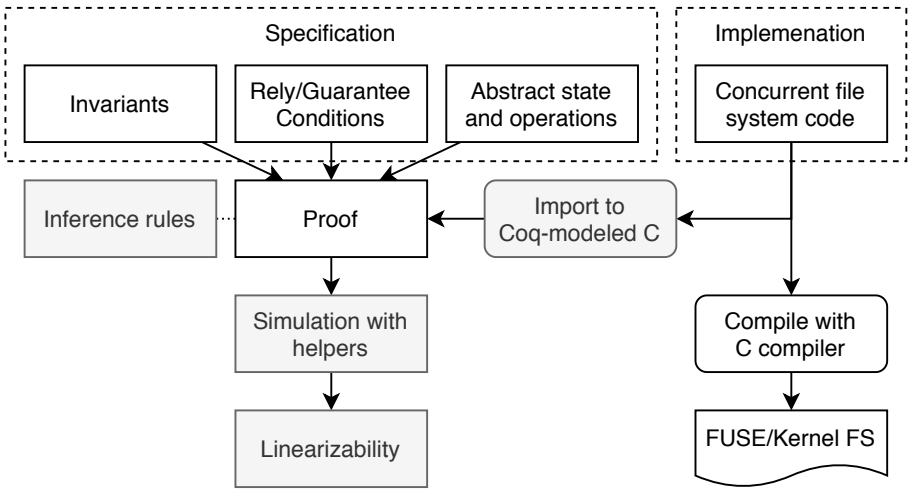

Figure 3.6: Development flow with CRL-H. Rectangular boxes denote source code; shaded boxed are components of the CRL-H framework; rounded boxes signify processes. The curved box at bottom right is the final FUSE executable with file system implementation. Note that the user-supplied C code is compiled directly without any intermediate translations; the Coq proof is *about* the C code. This diagram is taken from [62] (with minor modifications), where it was originally presented.

conditions, and invariants specifying interactions among threads. To prove that a given implementation contextually refines the specification, a forward simulation argument needs to be made, so the user has to present a suitable abstraction relation. CRL-H supplies two kinds of logic inference rules to facilitate this step: *C-statement-rules*, which allow the user to reason about effects of C code and evolution of the concrete state; and *implication-rules*, which allow the user to manipulate the abstract and ghost states in a manner suitable for showing the abstraction relation. In this sense, `linothers` is simply just an implication-rule where the user needs to supply a list of thread identifiers, and the state where all specified threads have been helped becomes the next abstract state. A notable advantage of CRL-H is its ability to analyze C code directly (see Figure 3.6), avoiding the need for any translation between a high-level, semantic-rich language (e.g. Coq's Gallina), and a low-level implementation language (such as C).

AtomFS

With all the theory developed in the previous subsections up until this point, AtomFS becomes more or less a showcase of CRL-H's abilities. AtomFS is an in-memory file system without a notion of true persistence (all changes are lost when the file system is unmounted), therefore its implementation is fairly simple. The proofs involving AtomFS maintain eight separate invariants.

One of the more interesting aspects of AtomFS's design is the *non-bypassable criterion*. This criterion states that with two threads competing to traverse the same file system path, an unhelped thread cannot pass a helped thread. The criterion, maintained as two separate invariants throughout the AtomFS's proofs, guarantees certain unlinearizable traces will not occur. Note that the criterion is stated in terms of helped threads, which is an abstract notion; in the actual C implementation, the same behavior is achieved by lock coupling, a technique where a thread will first acquire lock of the next inode in path before releasing lock of the previous inode. This coupling prevents one thread "passing" another one when traversing down the same path. Also note that this technique fails to work when file descriptor based operations are considered: for this reason, AtomFS will traverse the corresponding path even when a file descriptor is supplied to the operation.

CHAPTER 4
FUZZING EXPERIMENTS

This chapter focuses on fuzzing experiments with four file systems of our choice: FSCQ, Yxv6/Yggdrasil, Flashix and AtomFS. First, we describe our experimental setup in terms of used software tools, and present a novel test harness designed specifically for testing FUSE-based file systems. Challenges encountered during fuzzing of each of the file systems are presented next, along with results of the experiments. Finally, we discuss related work.

4.1 Experimental setup

4.1.1 AFL

American Fuzzy Lop (abbrev. AFL) is a security-oriented fuzzer by M. Zalewski [63]. Among other things, AFL relies on genetic algorithms to automatically discover new test cases, supports compile-time instrumentation (via a dedicated instrumenting LLVM IR pass), and features a fork-server mode (including an IPC protocol) to speed fuzzing up as much as possible. The last two features will be discussed next, as they are directly related to some of the practical challenges we faced.

In order to drive the genetic algorithms guiding new test input generation, AFL collects traces of test executions. Such traces allow AFL to determine e.g. if a given test input triggers a new control flow path in the target executable, which makes the input more valuable compared to other inputs. The traces are collected using a shared memory region mapped by both AFL and the target executable into their respective address spaces, see Figure 4.1. Prior to executing a test case, AFL wipes the shared memory to get rid of any previous trace. Then, the (instrumented) target binary in executed: the instrumentation code embedded in the binary maps the shared memory into address space of the test process

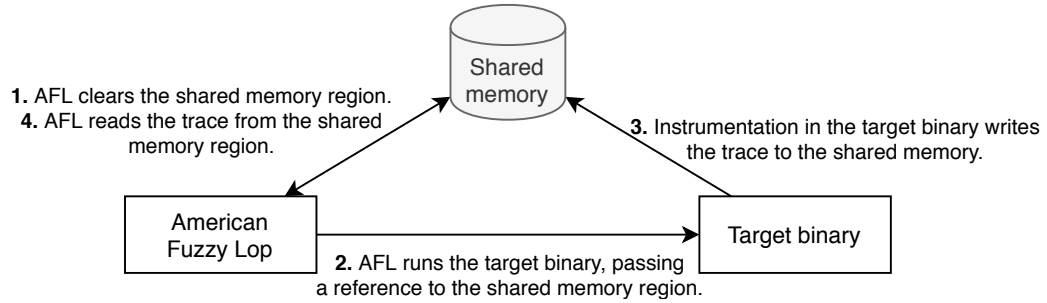

Figure 4.1: Collection of a run trace with AFL.

(a reference to the shared memory region is passed via an environment variable) and writes the trace into the shared memory. When the test process finishes, AFL simply reads the trace from the shared memory region.

Therefore, the instrumentation has two goals: first, it needs to map the shared memory region, and second, it needs to redirect the written trace to the mapped memory region. AFL offers multiple instrumentation options to achieve these goals; in this work, we rely on Clang- and LLVM-based instrumentation. AFL ships a wrapper around Clang named `afl-clang-fast`, which uses an LLVM IR pass for instrumentation of all compiled code at IR (intermediate representation) level, and links in supporting initialization code. The initialization code maps the shared memory region before the function `main` executes, employing a mechanism similar to how constructors of statically-allocated class objects are run in C++. The LLVM IR pass inserts code at the beginning of each basic block to write the trace to the mapped region via a global pointer variable.

The *fork-server mode* may be seen as an extension of the instrumentation. In addition to mapping the shared memory region, the initialization code can also use predefined file descriptors (by default, 198 and 199) to communicate with the parent AFL process and use `fork` to spawn new test processes directly, see Figure 4.2. Without a fork-server, AFL needs to call `fork` and `execve` to create a new test process every time, which is more costly. Moreover, AFL allows to combine the *fork-server mode* with a so-called *deferred mode*: in the deferred mode, the developer decides when the shared memory region is

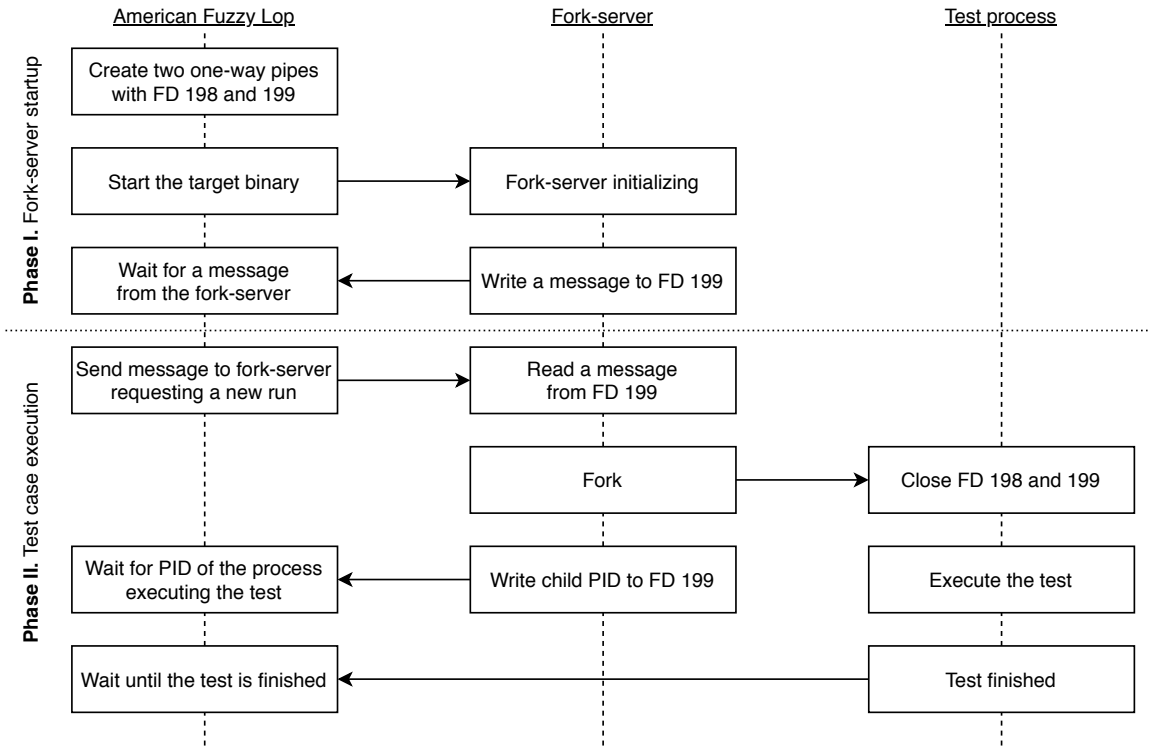

Figure 4.2: IPC protocol of AFL's fork-server.

mapped and the fork-server started by calling a designated initialization function. Starting the fork-server later in the `main` function gives the developer an opportunity to minimize the performance penalty imposed by expensive initialization that does not depend on fuzzed input; when the fork-server is started after the initialization is done, all children start out already initialized. However, taking advantage of the deferred mode requires changes to the source code of the target application.

4.1.2 Hydra / SymC3

Hydra (2019) is an extension of AFL introduced by S. Kim et al from Georgia Institute of Technology [9]. The sole focus of Hydra is fuzzing file systems: Hydra's test cases consist of so-called *syscall programs*, sequences of file system-related kernel calls meant to exercise the file system implementation. Hydra replaces all test input generation stages with two stages of its own: one that randomly mutates an existing syscall program, and one

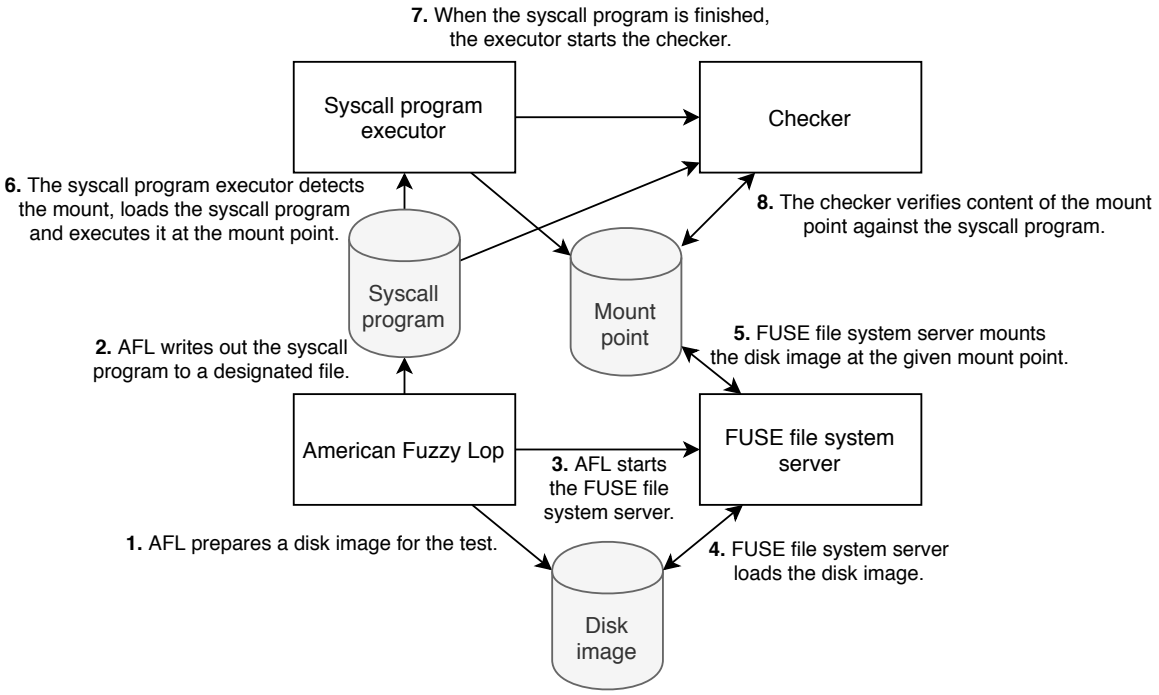

Figure 4.3: Testing a FUSE-based file system with Hydra.

which appends a new, randomly selected syscall.

While Hydra appears to primarily aim at fuzzing in-kernel file system implementations, it is also capable of fuzzing FUSE-based file systems; in fact, Kim et al report that Hydra was able to find bugs in FSCQ, one of the verified file system using FUSE. A simplified schema of a FUSE test case execution with Hydra is shown in Figure 4.3. After preparing a clean disk image and writing out the syscall program (which constitutes the test case) to a designated file, AFL starts the FUSE file system server to collect the execution trace. The server loads the disk image and mounts it at a pre-specified location. A *syscall program executor*, waiting in the background up until this point, detects the mount, reads the syscall program (written out by AFL earlier) from the designated file and executes it at the mount point. The desired AFL trace is collected during execution of the program. When the program is finished, the syscall program executor invokes a checker to validate the results. The checker can be an arbitrary program; Kim et al use SymC3, their own crash-consistency verifier acting as a reference interpretation of the syscall programs. SymC3 constructs the

expected content of the mount point after execution of the syscall program and compares it to the actual content of the mount point: if they match, the mount point has been validated.[1]

A fundamental limitation of the FUSE test design in Hydra is that AFL never receives the feedback from the checker, since the syscall program executor (and therefore the checker, too) is running *alongside* AFL. This is a missed opportunity: if AFL learned that a given syscall program causes a validation failure, it might be able to generate more interesting test cases.

4.1.3 FUSE test harness

One of the contributions of this thesis is a novel FUSE test harness, which represents a fresh look at the problem of testing a FUSE file system server. We have considered the following points to guide its design:

1. **FUSE server inputs.** The FUSE server under test has two inputs in general: a file system image and a sequence of syscalls operating on the file system. Fuzzing the file system image needs to be done in a file system-specific manner and requires non-trivial effort on the user's side. On the other hand, the syscalls have fairly well-defined meaning and are common to all file systems. Therefore, our FUSE test harness executes Hydra's syscall programs to exercise the FUSE server; the disk image is always a copy of an image supplied by the user. Note that the copy is unavoidable, since the FUSE server will mutate the image.

2. **Mount point validation.** Hydra's notion of external checkers makes sense: since the validation is not part of the harness itself, such design supports loose coupling of the components, permitting development of new checkers independently of the harness. However, the limitation of the Hydra's test design should be overcome; we must be able to supply the checker's feedback back to AFL. Therefore, the harness

[1] Note that this description deliberately glosses over some details, e.g. the FUSE server must be started twice to avoid polluting the AFL trace when validating the mount point.

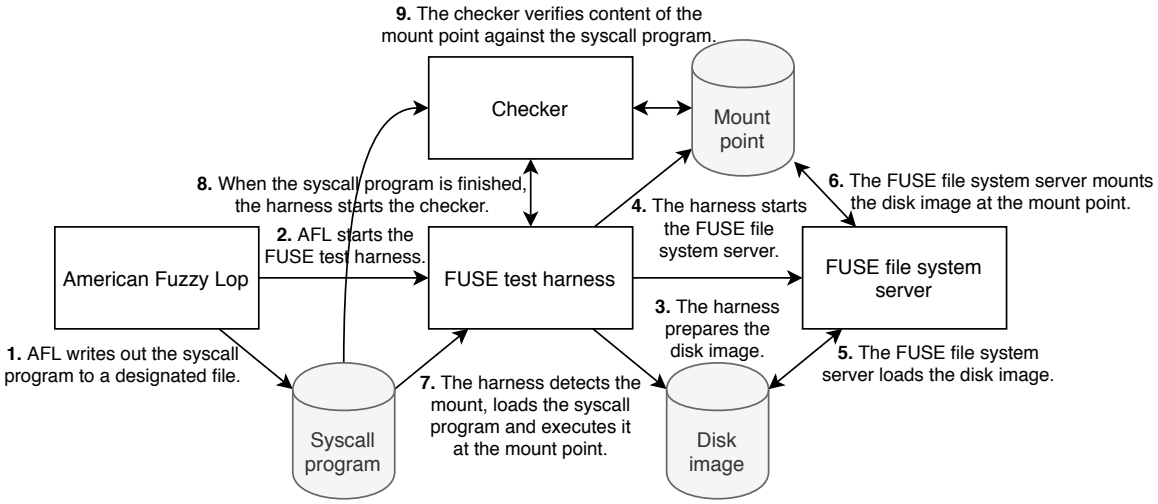

Figure 4.4: Testing a FUSE-based file system with the novel FUSE test harness.

necessarily has to communicate with AFL directly.

3. **Harness overhead.** Last but not least, any overhead imposed by the harness should be as minimal as possible; time spent on anything else but fuzzing is wasted.

Based on the considerations outlined above, we have implemented a revamped FUSE test design, displayed in Figure 4.4. Our design exhibits the following differences from the design of Kim et al:

1. **Communication with AFL.** The harness is invoked by AFL and communicates with AFL directly, which allows it to relay the checker feedback. The communication is achieved by linking in the AFL instrumentation initialization code, as described in Subsection 4.1.1. A fork-server is started after the harness is initialized; however, none of the code in the harness in actually instrumented, so the shared memory region for the AFL trace remains untouched. Note that the current implementation puts some restrictions on the instrumentation in the FUSE file server, namely the fork-server mode is not supported, and the instrumentation initialization code needs to be called manually. However, these restrictions are merely implementation artifacts and not technical limitations; both could be overcome with modest additional effort. We leave lifting these restrictions for future work.

2. **Disk image copy.** The harness, as opposed to AFL, assumes responsibility for preparing the test disk image. The harness implements the image copy by file cloning via `ioctl(FICLONE)`, if the underlying file system supports it; this approach has resulted in significant practical speedups on file systems which support Copy-on-Write mechanism for data blocks (e.g. Btrfs).

3. **Designated names.** The harness does not use designated names for any of the involved resources (i.e. the syscall program, the disk image or the mount point). Name of the syscall program file is passed from AFL on the command-line; the harness generates and manages unique temporary names for the disk image and the mount point, passing them to the FUSE file system server and the checker via command-line arguments as well. This design decision increases robustness of the harness.

4. **Syscall program execution.** The harness subsumes functionality of the syscall program executor, i.e. the syscall program is executed by the harness itself.

A detailed sequence diagram laying out the flow of a single test using the harness is given in Figure 4.5.

4.2 File system experiments

This thesis is focused exclusively on FUSE-based file systems. Reasons for this particular choice are two-fold: first, majority of the verified file system we have surveyed are built using FUSE, and second, some of the initial work has already been laid out by Kim et al.

Using our FUSE test harness reduces the developer's effort necessary to start fuzzing a new FUSE-based file system to the absolute minimum: all that is needed is building the file system with AFL instrumentation and calling the instrumentation initialization function before entering the main FUSE loop. Nevertheless, even just building with the instrumentation sometimes presents non-trivial challenges; we share our experience with instrumenting the selected file systems in the following subsections.

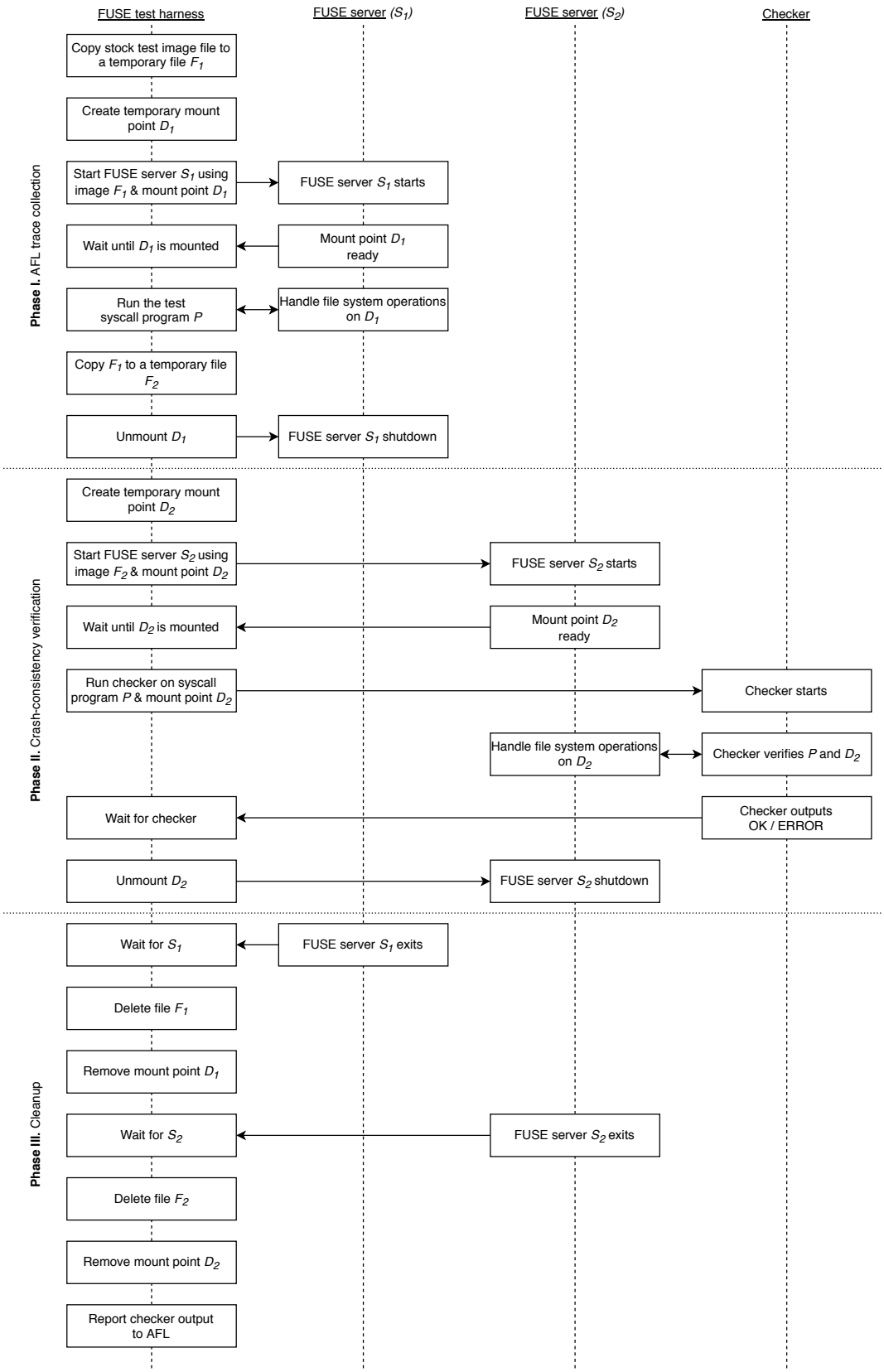

Figure 4.5: Flow of a FUSE server test.

We also run AFL with our FUSE test harness and SymC3 as a checker, and present the results of fuzzing each file system in several figures. We give test execution time breakdown with respect to growing syscall program size in order to pinpoint bottlenecks hindering greater test execution speeds. We present a plot of coverage and test execution time evolution over the duration of a single fuzzing session: such plot allows us to empirically infer if and how early AFL manages to saturate the coverage map, and how complicated the tests need to become to achieve such coverage. Finaly, we attempt to compare the FUSE test executor of Kim et al with our test harness in terms of time spent executing various stages of FSCQ tests.

4.2.1 FSCQ

Challenges

Since FSCQ has been successfully instrumented and fuzzed by Kim et al, we adopt their approach. After Haskell code is extracted from FSCQ's Coq source, it is compiled by GHC (Glasgow Haskell Compiler). AFL does not provide tools to instrument Haskell code; however, GHC supports LLVM as one of its code-generation back-ends. As a result, it is possible (via suitable command-line options) to have GHC use the instrumenting LLVM IR pass on the generated IR code, and link in the supporting initialization code. Moreover, the instrumentation initialization routine can be called via Haskell FFI interface from the entry-point function. In conclusion, a functional instrumented FSCQ binary can be obtained in more or less straightforward manner with modest effort.

Results

Figure 4.6 breaks down the execution time of a single FSCQ test. Total execution time grows with syscall program size, which is expected. For smaller programs, about 90–95 % of the execution time is split evenly between syscall program execution and checker

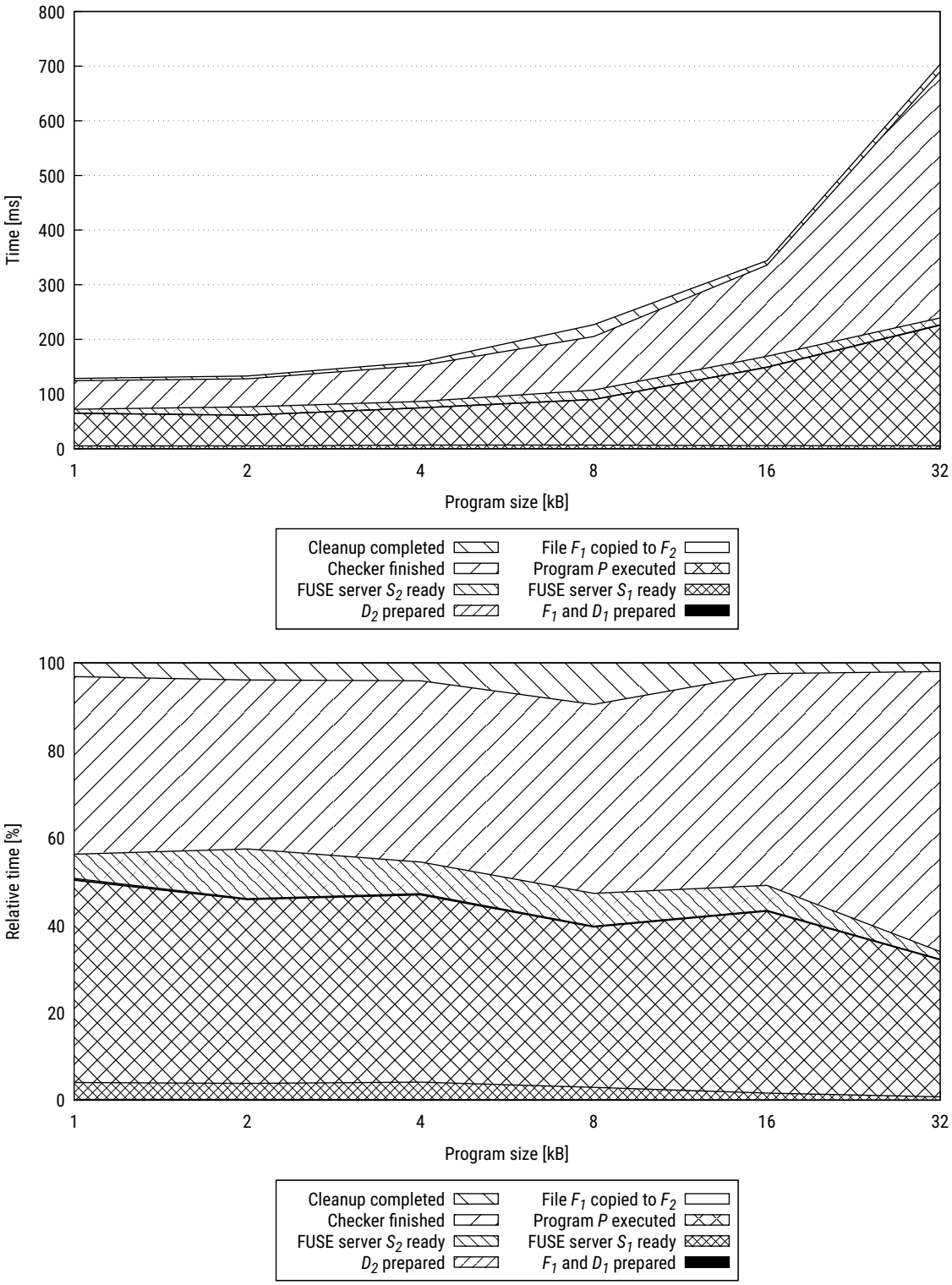

Figure 4.6: Absolute and relative execution time breakdown of FSCQ tests. The displayed run is one of five runs with the median total execution time. The legend refers to completion of various test stages; see Figure 4.5 for details.

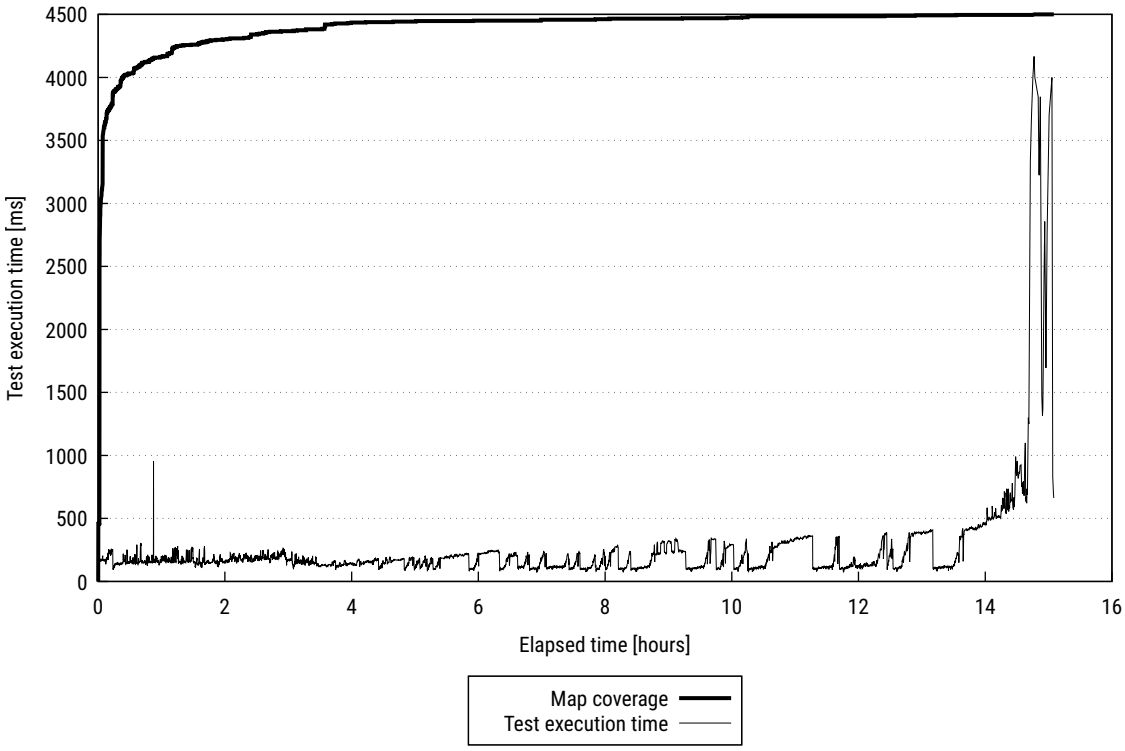

Figure 4.7: Evolution of coverage and test execution time over the duration of fuzzing FSCQ. Note that map coverage is unitless and does not relate to the time-labeled Y-axis.

execution, with the overhead imposed by FUSE server startup remaining minimal.[2] That being said, with growing program size there is a clear trend demonstrating domination of the total execution time by the checker, SymC3: with 32 kB syscall program, it consumes approximately 60% of the total execution time. We may conclude that optimizing SymC3 would be the best first step towards faster FSCQ tests with large syscall programs.

The progress of a single FSCQ fuzzing session is displayed in Figure 4.7. The map coverage appears to become (almost) saturated after about 4 hours. Note that after running for 14 hours, AFL decided to use more aggresive mutation, which resulted in the execution time spiking over 4 seconds. No bugs were found during the 15-hour fuzzing session.

Figure 4.8 tries to quantify some of the performance differences between Hydra's FUSE test executor and our FUSE test harness. We refrain from comparing the two more directly,

[2]This is in fact an impressive achievement of GHC, considering that Haskell is a high-level language, especially in contrast with other file systems presented later.

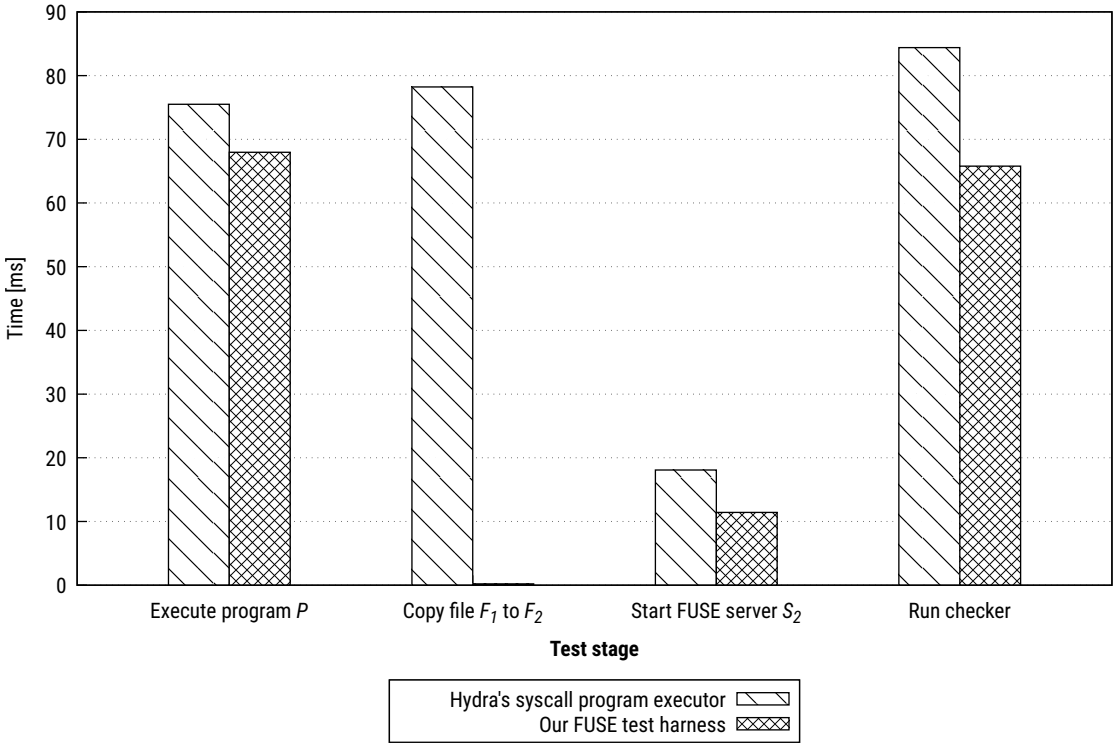

Figure 4.8: Comparison of Hydra's syscall program executor and our novel FUSE harness in terms of single test execution time. The measurements were done on a test run with 4 kB syscall program.

since they use a different test execution design (compare Figures 4.3 and 4.4). Nevertheless, we decided to at least compare the time spent executing the test stages present in both designs. Our harness manages to slightly improve on the performance of Hydra's test executor by relying on more efficient APIs (e.g. we use `fork` and `execve` to start the FUSE server and SymC3 processes, as opposed to `system`); most importantly, though, our harness practically eliminates the overhead of copying the disk image. This is achieved by cloning the file via `ioctl (FICLONE)`, which performs a constant-time file copy operation on file systems which support the copy-on-write mechanism, such as Btrfs. Kim et al recommend running their executor on tmpfs, which does not support this mechanism; as a result, copying the large disk images consumes a disproportionate amount of test execution time. This single optimization led to 2–5$\times$ faster test execution speeds, depending on program size.

4.2.2 Yxv6 / Yggdrasil

Challenges

Yxv6 is written in Python and relies on Cython to translate its Python code into C. The generated C sources are then compiled with a regular C compiler into native dynamic libraries, loadable as regular Python modules. Consequently, it is trivially possible to instrument Yxv6's code by using the C compiler wrapper provided by AFL, `afl-clang-fast`. Unfortunately, it is unclear how to link the instrumentation initialization code: it should not be linked into every module, but linking it into only one of the modules results in unresolved symbol errors in runtime. We were forced to rely on a workaround: we use `LD_PRELOAD` to force loading of the initialization code (linked into a dynamic library) before the Python executable or any of the Yxv6's modules. This workaround proved functional and enabled us to successfully fuzz Yxv6. The instrumentation initialization routine can be easily called via FFI provided by Cython.

Results

Execution time breakdown of Yxv6 tests is plotted in Figure 4.9. We may immediately observe that Yxv6 suffers from severe performance problems under larger workloads: syscall program execution time completely dominates all other stages for syscall programs bigger than 8 kB, even executing the checker.

Looking at Figure 4.10, the map coverage for Yxv6 becomes saturated within the first hour of fuzzing. We can see AFL going through several cycles before deciding to mutate more aggressively at about 7 hours into the fuzzing session, causing the test case execution time to exceed 6 seconds. The fuzzing session discovered a bug, which is described in Figure 4.11.

4.2.3 Flashix

Challenges

Flashix is an example of a file system which turned out to be quite difficult to instrument. To begin with, the Flashix FUSE server forks into background and does not exit when the file system is unmounted. While the latter is probably a bug, both of these issues prevent Flashix from being fuzzed with AFL. We were forced to manually investigate and debug these problems; luckily, we found an undocumented option to force the FUSE server to stay in foreground, solving both issues.

The public Flashix code is written in Scala, a functional language targeting the Java Virtual Machine (JVM); as such, the Scala compiler produces JVM byte-code. AFL provides no facilities for instrumenting Java byte-code; luckily, J. Judin has authored a toolkit named java-afl [64] for performing the same task as `afl-clang-fast`, except with JVM byte-code. The toolkit did not work out-of-the-box, possibly due to the fact that the byte-code was generated by Scala compiler, e.g. it has incorrectly detected more than one entry-point function and attempted to initialize the instrumentation more than once, leading to crashes.

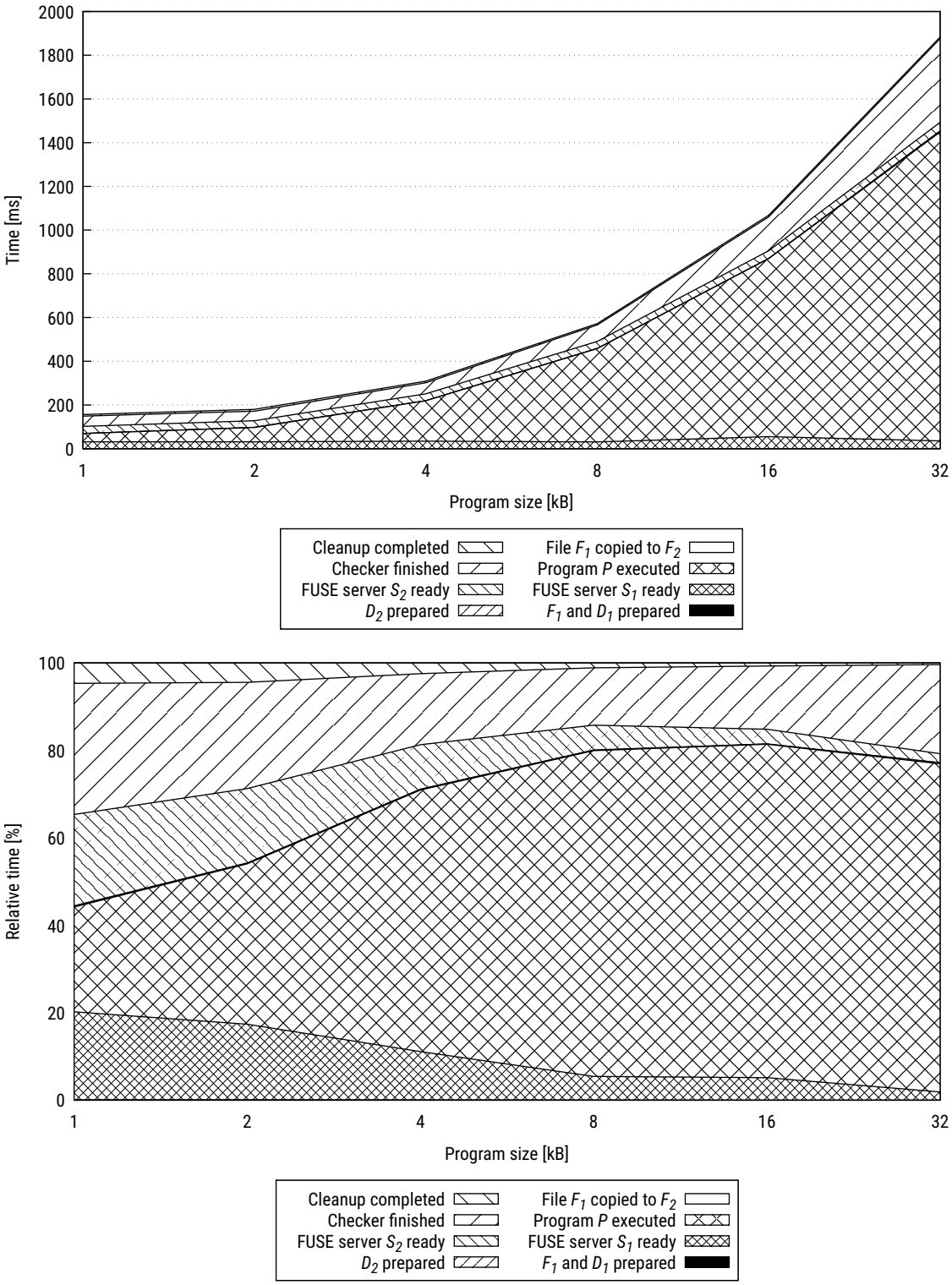

Figure 4.9: Absolute and relative execution time breakdown of Yxv6 tests. The displayed run is one of five runs with the median total execution time. The legend refers to completion of various test stages; see Figure 4.5 for details.

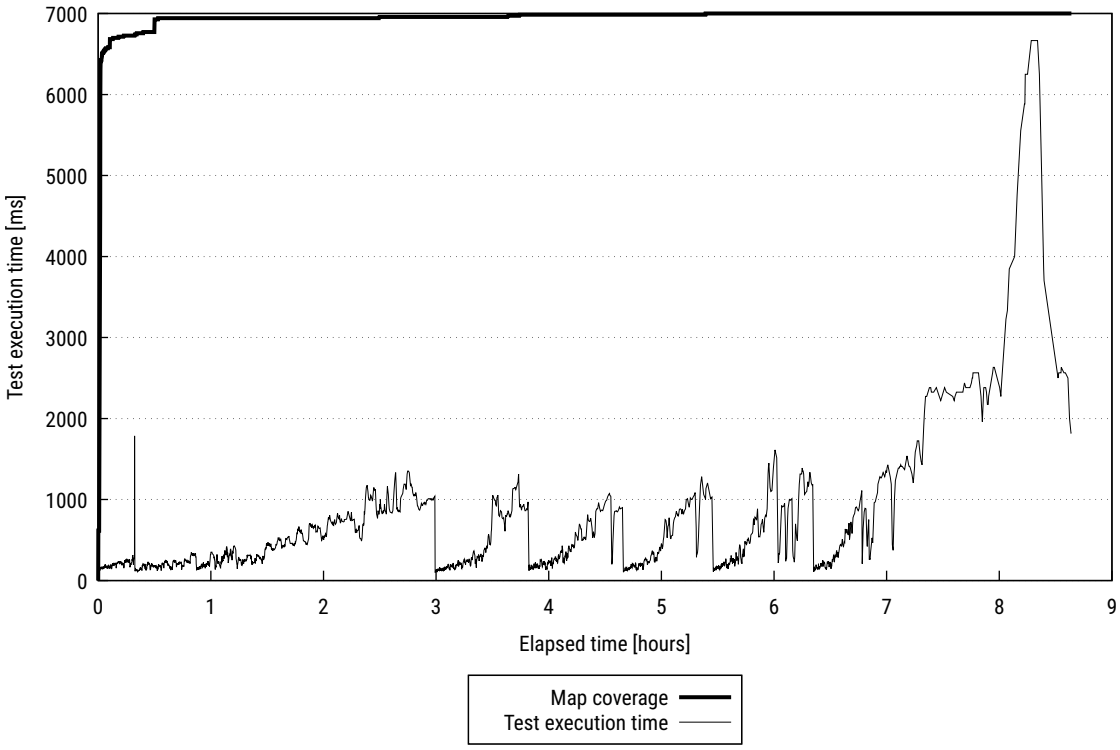

Figure 4.10: Evolution of coverage and test execution time over the duration of fuzzing Yxv6. Note that map coverage is unitless and does not relate to the time-labeled Y-axis.

```
$ truncate --size 256M disk-yxv6.img
$ mkdir -p mnt
$ python yav_xv6_main.py -o max_read=4096 \
> -o max_write=4096 -s mnt -- disk-yxv6.img &
$ cd mnt/
$ mkdir -p a/b c
$ python -c 'import os; os.rename ("a/b", "c")'
$ tree
.
|-- a
|-- c
`-- c

3 directories, 0 files
```

Figure 4.11: Listing from a terminal session demonstrating steps to reproduce a bug encountered in Yxv6 implementation. POSIX file system specification states that the directory `c` should have been replaced by `a/b`; however, Yxv6 lists both directories when enumerating contents of the file system root.

Eventually, however, we were able to successfully instrument the Flashix byte-code.

We encountered two more problems when testing Flashix with our FUSE harness. First, Flashix uses a hard-coded disk image file name, which required a simple adjustment to the source code. Second, certain operations fail after mounting the Flashix file system, e.g. appending data to a pre-existing file. Fortunately, we were able to discover a workaround: a single write-only operation (like creating a file or updating the last modification timestamp) on a freshly mounted file system corrects the problem. Therefore, our FUSE harness calls `utimes` on the mount point prior to running the syscall program when fuzzing Flashix.

The last and most persistent issue with fuzzing Flashix turned out to be JVM performance. In our experiments, the Java Virtual Machine took more than a second to start executing Flashix code. As shown in Figure 4.5, the FUSE server is actually invoked *twice* per one test case; altogether, a single Flashix test case required between 2.5–4 seconds to execute. Naturally, such execution rate severely limits what AFL can do. We attempted to improve performance of JVM in several ways:

1. **Persistent JVM.** Software called nailgun, developed by M. Lamb [65], allows to start only a single instance of JVM and reuse it for subsequent executions of the same Java byte-code. Unfortunately, it does not cooperate well with FUSE; in our experiments, the JVM process consistently crashed the second time we tried to mount a Flashix file system, printing a backtrace pointing to FUSE code.

2. **Ahead-of-time compilation.** JVM normally uses just-in-time compilation, translating the Java byte-code into native executable code in an on-demand fashion as the application runs. However, there is an option to compile the byte-code ahead-of-time with a separate compiler, avoiding the compilation penalty on JVM startup. We successfully compiled the base Scala library ahead-of-time, but observed no measurable speedup, and consequently abandoned this approach.

3. **Fork-server mode**. java-afl implements the AFL fork-server mode (see Subsection

4.1.1) in an effort to avoid the JVM startup penalty. The implementation comes with a warning, though: `fork` will only duplicate the calling thread, and since JVM uses many worker threads, it will not behave correctly. Our experiments proved the fork-server mode to be a non-option as well; the JVM simply got stuck and hanged as soon as it tried to do any disk I/O.

In the end, we were unable to achieve a significant speedup of Flashix tests, and proceeded to fuzz Flashix even with the extremely low execution rate.

Results

Figure 4.12 gives hard data to support our hypothesis that FUSE server startup is by far the biggest bottleneck in Flashix test executions. A single JVM initialization takes over a second, which, considering the FUSE server needs to be started twice, adds almost 2.5 seconds to every test case execution. The startup overhead is constant, though, so for larger syscall programs its role starts to diminish. At that point, however, the total test execution time is approaching 5 seconds.

Questions about effects of low execution rate on the fuzzing process are answered by Figure 4.13. Note that Flashix is the only file system which we fuzzed in parallel with three AFL instances (one master and two slaves). The immediately striking fact about the plot is that the map coverage does not seem to be nearing saturation even after 60 hours of continuous parallel fuzzing. This is certainly a limitation with regards to what bugs could have been discovered; during the 60-hour long fuzzing session, we detected none. Note that the spiking execution times around hour 15 and 62 are merely benign artifacts, caused by unrelated transient load on the fuzzing system.

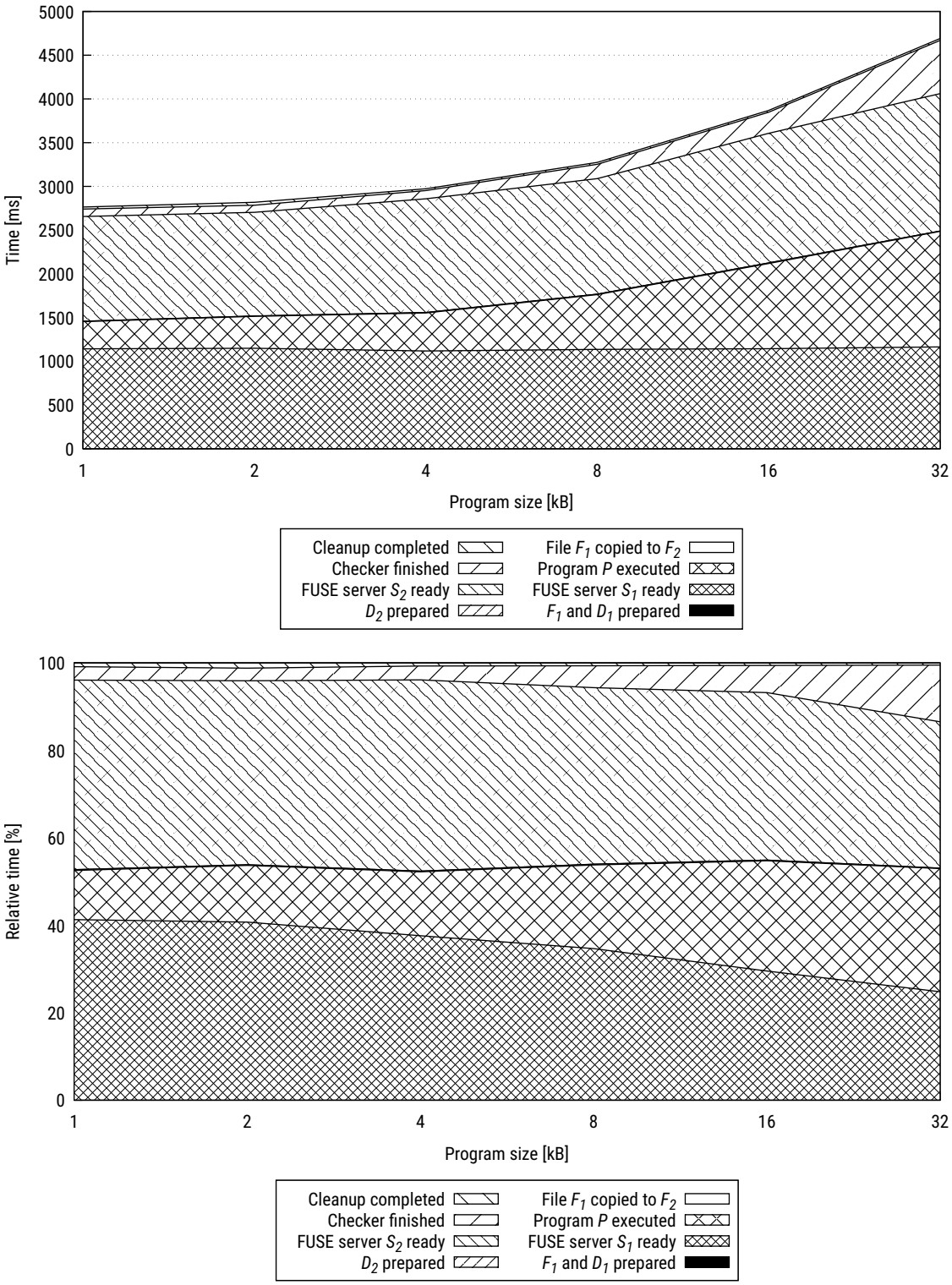

Figure 4.12: Absolute and relative execution time breakdown of Flashix tests. The displayed run is one of five runs with the median total execution time. The legend refers to completion of various test stages; see Figure 4.5 for details.

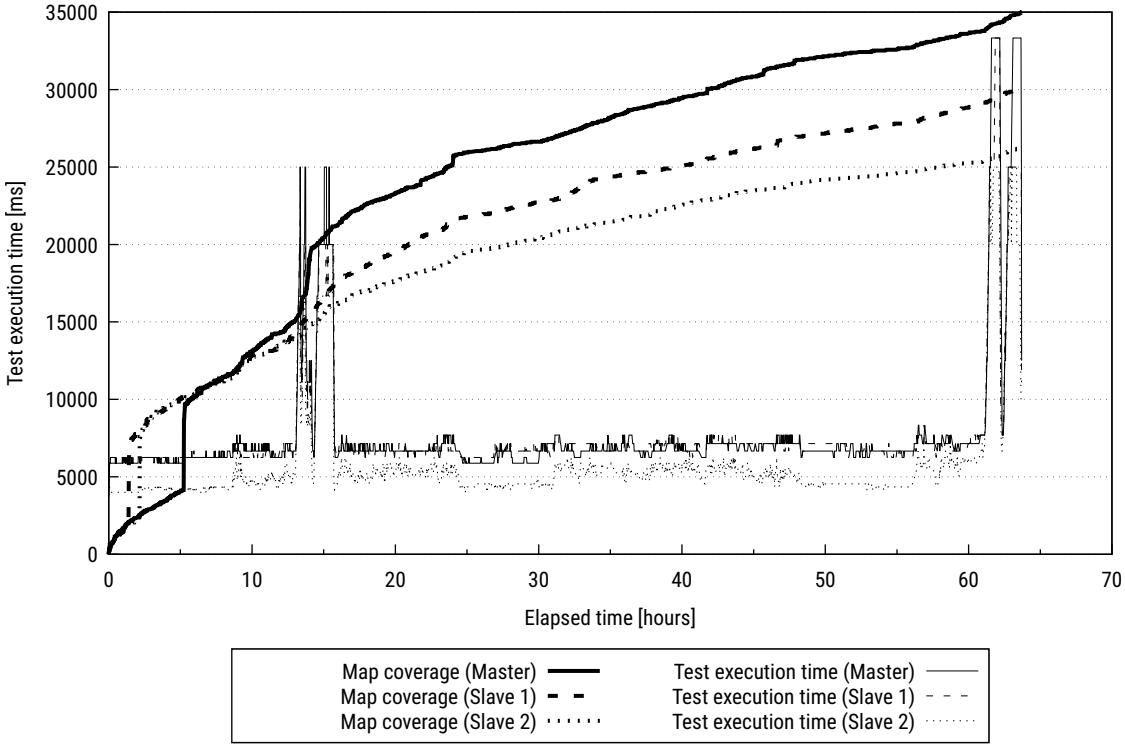

Figure 4.13: Evolution of coverage and test execution time over the duration of fuzzing Flashix. Note that map coverage is unitless and does not relate to the time-labeled Y-axis.

4.2.4 AtomFS

Challenges

AtomFS is implemented in C, so `afl-clang-fast` is able to instrument the source out-of-the-box. Calling the instrumentation initialization function is similarly easy, the function just needs to be declared with the correct signature.

As noted in Chapter 3, AtomFS is an in-memory file system and maintains no notion of persistence, so it effectively ignores the disk image file (even though one is still required). This proved to be a problem, since syscall programs need a non-empty file system; we worked around this particular issue by having the FUSE test harness re-create the expected file system content before executing the syscall program. Also, we could not remount the file system before the checker is run, so we decided to run the checker on the testing mount (D_1 is Figure 4.5); the checker introduces some noise into the collected AFL trace, but that was deemed acceptable.

Finally, note that the main contribution of AtomFS is verification of concurrency. Unfortunately, neither Hydra nor SymC3 provide any provisions for concurrent executions: Hydra's test case consists of only a single syscall program, and SymC3 cannot reason about concurrent executions. Properly validating concurrent executions is a good subject of future work.

Results

AtomFS is the only file system whose hand-written implementation is in C, and the plots in Figure 4.14 definitely give a testament to that: the checker, SymC3, is the critical bottleneck in AtomFS tests, consuming over 80 % of the total execution time. As noted in the previous subsection, some stages (namely disk image copy and start of the second FUSE server instance) do not apply in AtomFS tests, and are plotted as taking no time.

Figure 4.15 displays the fuzzing session of AtomFS. There is not much to see; the

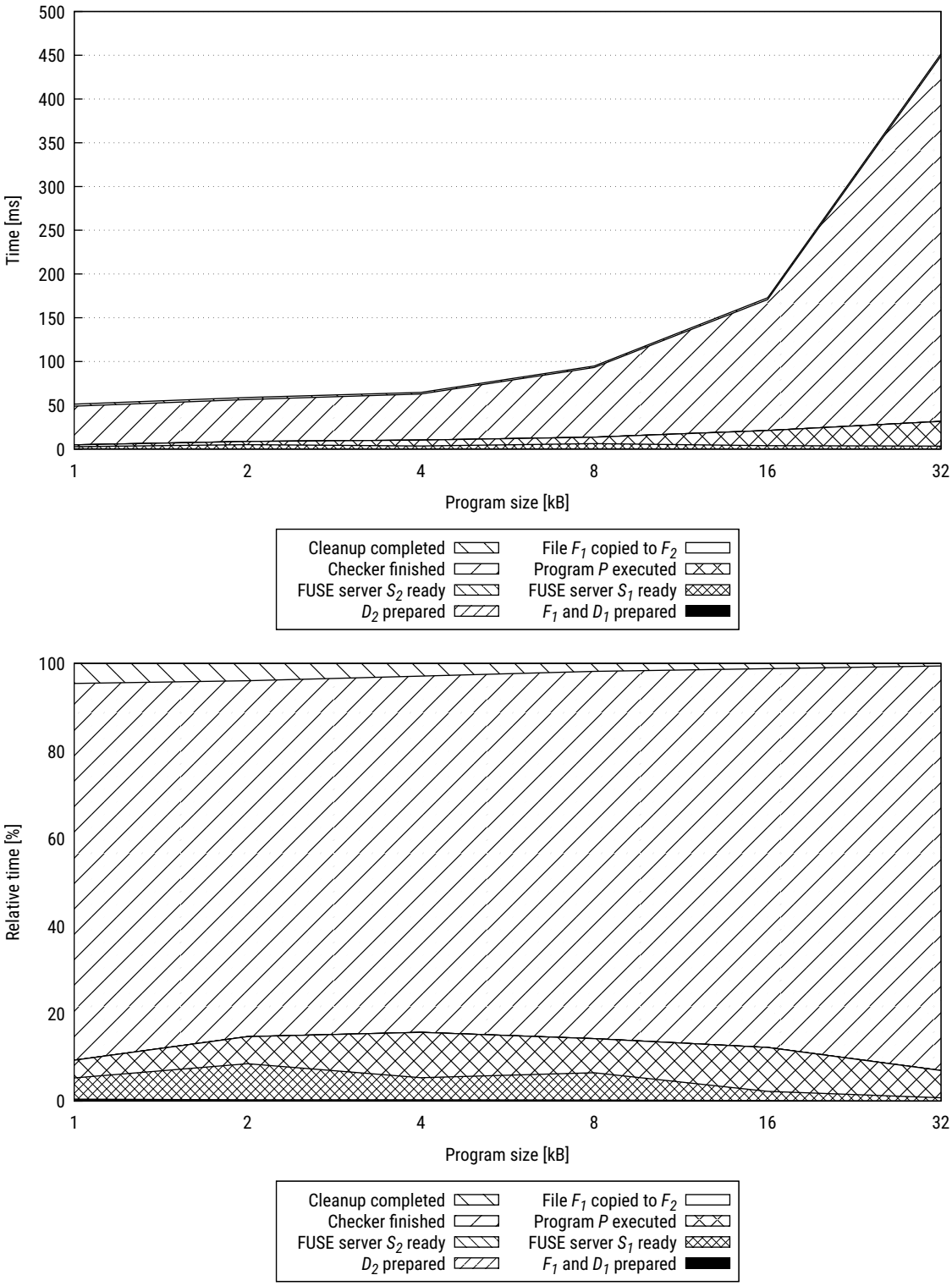

Figure 4.14: Absolute and relative execution time breakdown of AtomFS tests. The displayed run is one of five runs with the median total execution time. The legend refers to completion of various test stages; see Figure 4.5 for details.

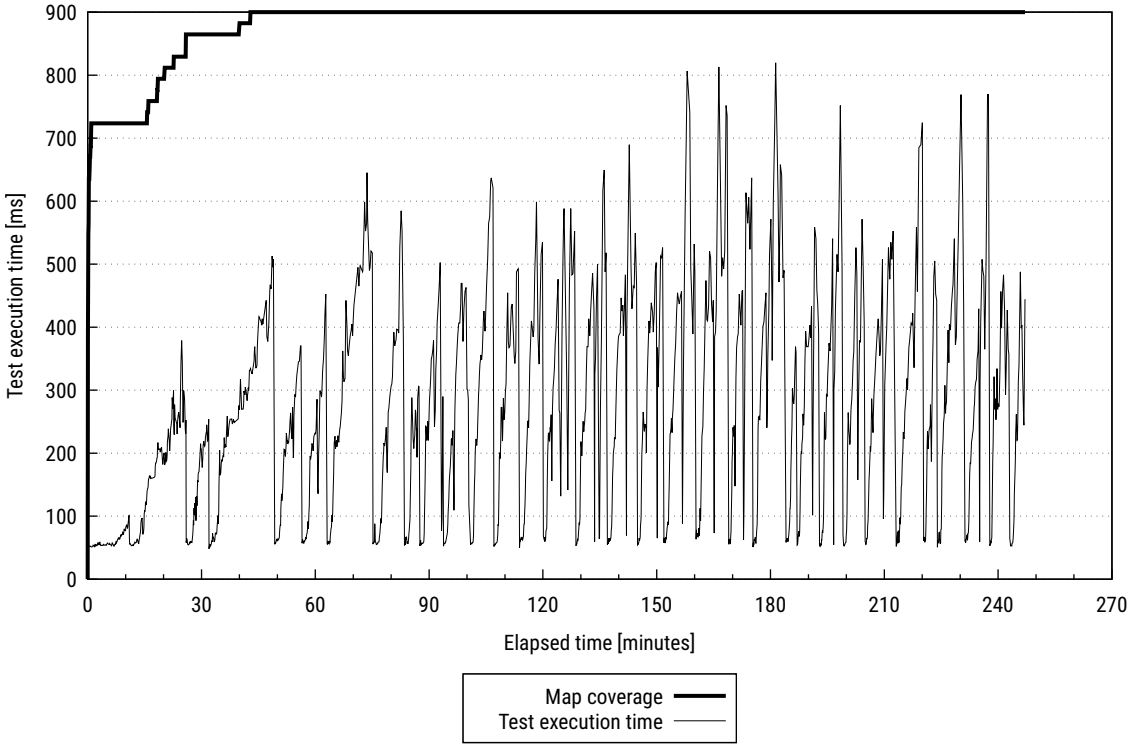

Figure 4.15: Evolution of coverage and test execution time over the duration of fuzzing AtomFS. Note that map coverage is unitless and does not relate to the time-labeled Y-axis.

map coverage becomes saturated within 60 minutes of starting the fuzzing process, and the following "comb" formed by spiking execution times is an evidence to how AFL spins in cycles, struggling to increase coverage. The fuzzing session discovered no bugs in AtomFS.

CHAPTER 5
CONCLUSION

In this work, we introduced and briefly discussed some of the concepts and methods used in formal reasoning and verification of computer programs; we surveyed nearly twenty software projects (focusing predominantly on systems software) which use formal verification for extended correctness guarantees, and outlined their high-level approach to applying formal methods in practice; we selected four verified file systems (FSCQ, Yxv6 / Yggdrasil, Flashix and AtomFS) for empirical validation of their correctness via software fuzzing; we presented our experimental fuzzing setup, based on American Fuzzy Lop and prior work of S. Kim et al on fuzzing file systems, introducing a novel test harness designed specifically for fuzzing FUSE-based file systems; and finally, we presented results of the fuzzing sessions with each file system.

We discovered what we believe to be a bug in only one of the file systems, namely Yxv6. While this result may seem encouraging from the perspective of formal verification, software fuzzing as a technique suffers from the same flaw as plain software testing: we may never be absolutely sure if our result is yet another certificate of the file systems' correctness, or evidence of our fuzzing experiments' limited abilities. In Chapter 4, we specifically note that in at least two cases the experiments were unable to fully exercise the file system under test: namely, the Flashix tests suffered from severe performance issues due to prolonged startup time of the Java Virtual Machine, and the tests of AtomFS could not verify its behavior under concurrent loads due to inherent limitations of Hydra and SymC3. We suggest that a suitable direction for future work in this area might be extending Hydra and SymC3 with the ability to generate and validate parallel file system test cases.

REFERENCES

[1] X. Leroy, "A formally verified compiler back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, p. 363, 2009.

[2] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 1, pp. 1–70, 2014.

[3] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, "An empirical study on the correctness of formally verified distributed systems," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 328–343.

[4] L. Lamport, "Specifying concurrent systems with TLA$^+$," *NATO ASI Series F: Computer and Systems Sciences*, vol. 173, pp. 183–250, 1999.

[5] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[6] Inria, CNRS, *et al.* (1999–2020). Reference manual of the Coq proof assistant, [Online]. Available: `https://coq.inria.fr/refman/`.

[7] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[8] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.

[9] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding semantic bugs in file systems with an extensible fuzzing framework," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 147–161.

[10] M. Ben-Ari, *Mathematical Logic for Computer Science*, 3rd ed.. Springer Science & Business Media, 2012, ISBN: 1-4471-4129-6.

[11] D. Miller, "Logic: Higher-order," *Encyclopedia of Artificial Intelligence, 2nd ed.*, 1991. [Online]. Available: `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/encyclopedia.pdf`.

[12] N. Bezhanishvili, D. de Jongh, *et al.*, *Intuitionistic logic*. Institute for Logic, Language and Computation (ILLC), University of Amsterdam, 2006.

[13] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[14] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, IEEE, 2002, pp. 55–74.

[15] R. Nederpelt and H. Geuvers, *Type theory and formal proof: an introduction*. Cambridge University Press, 2014.

[16] A. Church, "An unsolvable problem of elementary number theory," *American journal of mathematics*, vol. 58, no. 2, pp. 345–363, 1936.

[17] H. R. Lewis and C. H. Papadimitriou, "Elements of the theory of computation," *ACM SIGACT News*, vol. 29, no. 3, pp. 62–78, 1998.

[18] A. Church, "A formulation of the simple theory of types," *The journal of symbolic logic*, vol. 5, no. 2, pp. 56–68, 1940.

[19] J.-Y. Girard, "The system f of variable types, fifteen years later," *Theoretical computer science*, vol. 45, pp. 159–192, 1986.

[20] J. B. Wells, "Typability and type checking in the second-order λ-calculus are equivalent and undecidable," in *Proceedings Ninth Annual IEEE Symposium on Logic In Computer Science*, IEEE, 1994, pp. 176–185.

[21] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1982, pp. 207–212.

[22] L. Jutting, "Typing in pure type systems," *Information and Computation*, vol. 105, no. 1, pp. 30–41, 1993.

[23] S. Blazy, Z. Dargaye, and X. Leroy, "Formal verification of a C compiler front-end," in *International Symposium on Formal Methods*, Springer, 2006, pp. 460–475.

[24] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: A verified implementation of ML," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 179–191, 2014.

[25] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish, "The verified CakeML compiler backend," *Journal of Functional Programming*, vol. 29, 2019.

[26] A. Lööw, R. Kumar, Y. K. Tan, M. O. Myreen, M. Norrish, O. Abrahamsson, and A. Fox, "Verified compilation on a verified processor," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 1041–1053.

[27] R. Kumar, "Self-compilation and self-verification," University of Cambridge, Computer Laboratory, Tech. Rep., 2016.

[28] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad apps: End-to-end security via automated full-system verification," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 165–181.

[29] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 917–934.

[30] A. Fromherz, N. Giannarakis, C. Hawblitzel, B. Parno, A. Rastogi, and N. Swamy, "A verified, efficient embedding of a verifiable assembly language," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[31] T. Chajed, F. Kaashoek, B. Lampson, and N. Zeldovich, "Verifying concurrent software using movers in CSPEC," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 306–322.

[32] T. Chajed, "Verifying an I/O-concurrent file system," PhD thesis, Massachusetts Institute of Technology, 2017.

[33] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, *et al.*, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.

[34] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo, "CertiKOS: A certified kernel for secure cloud computing," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, 2011, pp. 1–5.

[35] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo, "Deep specifications and certified abstraction layers," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 595–608, 2015.

[36] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "CertiKOS: An extensible architecture for building certified concurrent OS kernels," in

12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016, pp. 653–669.

[37] R. Gu, Z. Shao, J. Kim, X. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. Ramananandro, "Certified concurrent abstraction layers," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 646–661, 2018.

[38] R. Gu, Z. Shao, H. Chen, J. Kim, J. Koenig, X. Wu, V. Sjöberg, and D. Costanzo, "Building certified concurrent OS kernels," *Communications of the ACM*, vol. 62, no. 10, pp. 89–99, 2019.

[39] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang, "Hyperkernel: Push-button verification of an OS kernel," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 252–269.

[40] D. Johnson, "Porting hyperkernel to the ARM architecture," Technical Report UW-CSE-17-08-02. University of Washington, Tech. Rep., 2017.

[41] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang, "Push-button verification of file systems via crash refinement," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 1–16.

[42] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 287–305.

[43] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. Wang, "Nickel: A framework for design and verification of information flow control systems," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 287–305.

[44] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," *Communications of the ACM*, vol. 54, no. 11, pp. 93–101, 2011.

[45] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, "Scaling symbolic evaluation for automated verification of systems code with Serval," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 225–242.

[46] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 357–368.

[47] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "IronFleet: Proving practical distributed systems correct," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 1–17.

[48] M. Lesani, C. J. Bell, and A. Chlipala, "Chapar: Certified causally consistent distributed key-value stores," *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 357–370, 2016.

[49] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, *et al.*, "Cogent: Verifying high-assurance file system implementations," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 175–188, 2016.

[50] H. Chen, D. Ziegler, A. Chlipala, M. F. Kaashoek, E. Kohler, and N. Zeldovich, "Specifying crash safety for storage systems," in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[51] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using Crash Hoare logic for certifying the FSCQ file system," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 18–37.

[52] H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Verifying a high-performance crash-safe file system using a tree specification," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 270–286.

[53] A. Ileri, T. Chajed, A. Chlipala, F. Kaashoek, and N. Zeldovich, "Proving confidentiality in a file system using DiskSec," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 323–338.

[54] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif, "A formal model of a virtual filesystem switch," *arXiv preprint arXiv:1211.6187*, 2012.

[55] ——, "Verification of a virtual filesystem switch," in *Working Conference on Verified Software: Theories, Tools, and Experiments*, Springer, 2013, pp. 242–261.

[56] J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif, "Crash-safe refinement for a verified flash file system," 2014.

[57] G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif, "Development of a verified flash file system," in *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, Springer, 2014, pp. 9–24.

[58] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif, "Inside a verified flash file system: Transactions and garbage collection," in *VSSTE*, Springer, 2015, pp. 73–93.

[59] G. Ernst, "A verified POSIX-compliant flash file system-modular verification technology & crash tolerance," 2017.

[60] Y. Gurevich *et al.*, *Evolving algebras 1993: Lipari guide.* 1993.

[61] E. Börger, "The abstract state machines method for high-level system design and analysis," in *Formal Methods: State of the Art and New Directions*, Springer, 2010, pp. 79–116.

[62] M. Zou, H. Ding, D. Du, M. Fu, R. Gu, and H. Chen, "Using concurrent relational logic with helpers for verifying the AtomFS file system," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 259–274.

[63] M. Zalewski. (2019). American fuzzy lop, [Online]. Available: `https://lcamtuf.coredump.cx/afl/`.

[64] J. Judin. (2018). java-afl: Binary rewriting approach with fork server support to fuzz java applications with afl-fuzz, [Online]. Available: `https://github.com/Barro/java-afl`.

[65] M. Lamb. (2017). Nailgun: Insanely fast java, [Online]. Available: `http://martiansoftware.com/nailgun/`.